In AAAI 2020 Workshop on Generalization in Planning (GenPlan 2020)
New York City, NY, USA

Learning Generalized Models by Interrogating Black-Box Autonomous Agents

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

Abstract

This paper develops a new approach for estimating a relational model of a non-stationary black-box autonomous agent that can plan and act. In this approach, the user may ask an autonomous agent a series of questions, which the agent answers truthfully. Our main contribution is an algorithm that generates an interrogation policy in the form of a contingent sequence of questions to be posed to the agent. Answers to these questions are used to derive a minimal, functionally indistinguishable class of agent models. This approach requires a minimal query-answering capability from the agent. Empirical evaluation of our approach shows that despite the intractable space of possible models, our approach allows correct and scalable estimation of relational STRIPS-like agent models for a class of black-box autonomous agents.

1 Introduction

Growing deployment of autonomous agents leads to a pervasive problem: how would we ascertain whether an autonomous agent 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; and they may adapt and learn from the environment where they are deployed, invalidating design-stage knowledge of agent models.

Such scenarios feature two properties that limit the applicability of existing approaches for model learning: they feature (a) *non-stationary* agents and environments, which evolve with the situation and do not present sufficient observational data for pure statistical learning and (b) *black-box* autonomous agents that may not be aware of a user's preferred model representation.

This paper presents a new approach for estimating an interpretable, relational model of a black-box autonomous agent that can plan and act, by interrogating it. Our approach is inspired by the interview process that one typically uses to determine whether a human would be qualified for a given task. Consider a situation where Hari(ette) ($\mathcal{H}$) wants their autonomous 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 adaptive, non-stationary black-box agents by interrogating them, thus improving the overall usability as well as field-debuggability of autonomous systems.

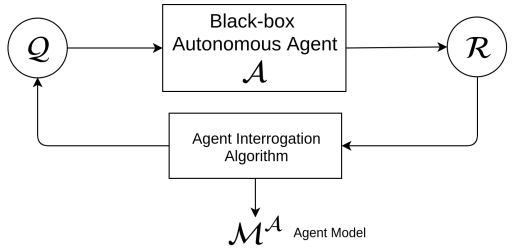

Figure 1: The agent interrogation framework

We use a rudimentary class of queries that makes our approach applicable on 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 hypothetical plan π on a hypothetical initial state s. These are the only queries that $\mathcal{A}$ needs to support for our approach to be applicable. Fig. 1 illustrates the overall paradigm. Our approach generates a sequence of queries ($\mathcal{Q}$) depending on the agent's responses ($\mathcal{R}$) during the query process; the result of the overall interrogation process is a complete model of $\mathcal{A}$. In order to generate queries, we present 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 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 as-

[1]Just 2 actions and 5 grounded propositions would yield $9^{2\times5} \sim 10^9$ possible STRIPS models – each proposition could be absent, positive or negative in the precondition and effects of each action. 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.

sume 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 always answers truthfully.

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 non-stationary 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 observation traces. Aineto et al. [2019] present an approach for reducing model recognition to planning. Their approach is shown to work even for partially observed agent behaviors. 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 using plan traces. LOUGA (Kučera and Barták 2018) uses a combination of genetic algorithm and an ad-hoc method to learn planning operators using plan traces. 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. While these prior approaches address the problem of learning models from training data, they do not address the problem of estimating agent models in situations where the agent or the environment are not stationary – i.e., where the agent may learn, where it may be provided with system updates, or where the environment changes such that the distribution that the training data came from is no longer representative of the deployment.

In contrast, our approach is not limited to stationary settings since it acquires current information using auto-generated queries. Moreover, our approach does not require $\mathcal{A}$ to provide intermediate states in an execution, or to have an Oracle that could provide counterexamples or assess the correctness of the current model estimate. 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. In contrast, our approach uses a 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 when such information is available.

The rest of this paper is organized as follows. The next section presents some background terminology used in this paper and explains our approach. Section 3 discusses the empirical evaluation of our approach; and Section 4 highlights our main conclusions and directions for future work.

2 The Agent-Interrogation Task

We assume that $\mathcal{H}$ needs to estimate $\mathcal{A}$'s model using 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 state variables, each with a finite domain $dom(p_i)$ associated with it; $\mathbb{A} = \{a_1, \ldots, a_k\}$ is a finite set of actions, each represented as a tuple $\langle pre(a_j), \textit{eff}(a_j) \rangle$, where $pre(a_j)$ and $\textit{eff}(a_j)$ are the partial assignments of the state variables. While we explain the salient points of this representation and its grounded, propositional form as needed, we refer the reader to prior work for formal descriptions (Helmert 2009).

The only information $\mathcal{H}$ has is the set of actions $\mathbb{A}$ that $\mathcal{A}$ can perform. As noted in the Introduction, $\mathcal{A}$ has functional definitions of the predicates 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 $\langle \mathcal{M}^{\mathcal{A}}, \mathbb{Q} \rangle$ where $\mathcal{M}^{\mathcal{A}}$ is the true model of the agent (unknown to the interrogator) being interrogated, and $\mathbb{Q}$ is the class of queries that can be posed to the agent by the interrogator. 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. *Solutions to the agent interrogation task* take the form of a query policy $\theta^* \rightarrow \mathbb{Q} \cup \{\textit{Stop}\}$ that maps 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*.

Although our approach is developed for relational representations, we will utilize a propositional representation for ease of exposition in most of this paper; we note specific salient points about relational representations when they are not clear from context. We evaluate our approach on relational models (Sec. 3). In the propositional form, we assume that the common vocabulary consists of a grounded set of actions $\mathbb{A}^G$ and a set of propositions $\mathbb{P}$.

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 a proposition from the common vocabulary of propositions $\mathbb{P}$; a is a grounded action from the set of grounded actions $\mathbb{A}^G$, $l \in \{\textit{pre}, \textit{eff}\}$ 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 proposition λ_p appears in an action λ_a at

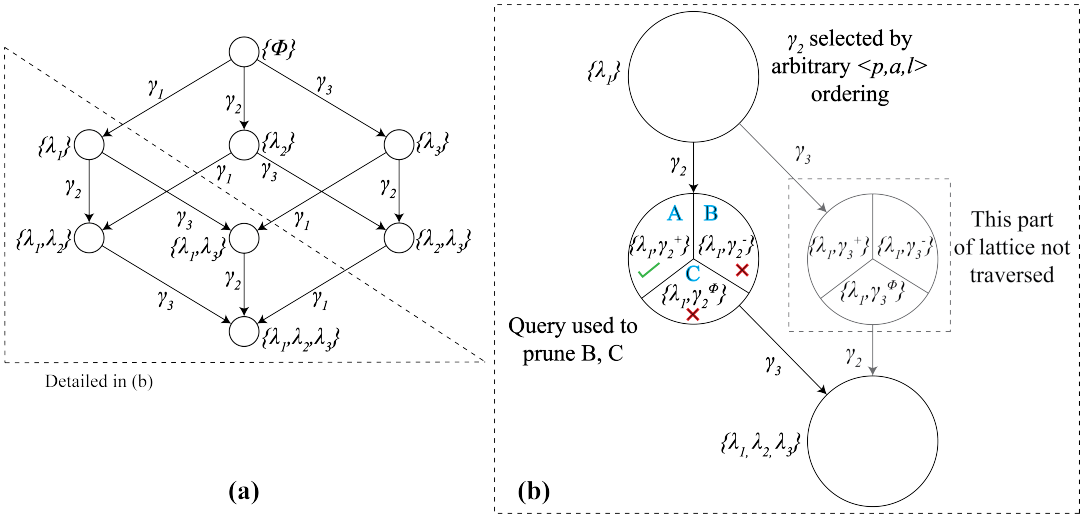

Figure 2: (a) A subset lattice created using pal-tuples; (b) Detailed view of a portion of lattice (marked in (a)) illustrating how partitions are created and pruned.

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.

While applying this approach to relational models, we use action arguments to define the set of atoms that can take the place of p in palm tuples. E.g., palm tuples involving an action $a(x, y)$ will be of the form $\langle p(var_1, var_2), a(x, y), l, m\rangle$, where var_1 and var_2 take unequal values from the action arguments $\{x, y\}$, and l and m have the same domains as in the propositional case above. Since this notation can be unnecessarily tedious, we use the more succinct propositional representation given above for notational convenience and implement the relational version in all our experiments. Also note here that for the example given above, $\mathcal{H}$ can form two variations of the predicate p; $p(x, y)$ and $p(y, x)$ as it has no knowledge of the semantics of the domain. We use $|\mathbb{P}^*|$ to represent the number of all such predicates and it depends on $|\mathbb{P}|$ and maximum arities of largest predicate and action.

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 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 λ tuple.

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 $f_\lambda : \mathcal{M} \mapsto (\mathcal{M} \setminus \lambda)$, where $f_\lambda : \mathcal{U} \to \mathcal{U}$. A set 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. 2(a)). Note that at each node we can have all possible variants of a palm tuple. For example the topmost node in Fig. 2(b), 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 version of the models and we get more concretized models as we move down.

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 $\Lambda = \Gamma \times \{+, -, \emptyset\}$, 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. 2(b), 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 $\mathcal{Q}$. Also, in these model lattices every node has an edge going out of 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 each lattice node.

Form of Agent Queries As discussed earlier, we pose queries to the agent and based on the responses we try to infer the agent's model. We express queries as functions mapping models to answers. More precisely, let $\mathcal{U}$ be the set of possible models and $\mathcal{R}$ a set of possible responses. A *query* $\mathcal{Q}$ is a function $\mathcal{Q} : \mathcal{U} \rightarrow \mathcal{R}$.

In this paper we utilize only one class of queries: *plan outcome queries*, which are parameterized by a state $s_{\mathcal{I}}$ and a plan π.

Plan outcome queries ($\mathcal{Q}_{PO}$) ask the agent the length of the longest prefix of the plan π that can be executed successfully when starting in the state $s_{\mathcal{I}} \in 2^{\mathbb{P}}$, and the resulting final state. E.g., "Given that the bottles *b1* and *b2* with labels *l1* and *l2* respectively are kept on the floor and your hand is empty, what would happen if you executed pickup(b1), pickup(b2), place(b1, s1), place(b2, s1)?

A response to these queries can be of the form "I can execute the plan till step ℓ and at the end of it *b1* is on shelf *s1* and *b2* is on shelf *s1*". 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 $s_{\mathcal{F}} \in 2^{\mathbb{P}}$ 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 $s_{\mathcal{F}} \in 2^{\mathbb{P}}$ 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, $\mathcal{Q}_{PO} : \mathcal{U} \rightarrow \mathbb{N} \times 2^{\mathbb{P}}$, where $\mathbb{N}$ is the set of natural 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 $\mathcal{M}_i \rceil\lfloor^{\mathcal{Q}} \mathcal{M}_j$, iff $\mathcal{Q}(\mathcal{M}_i) \neq \mathcal{Q}(\mathcal{M}_j)$.

Given a pair of abstract models, we wish to determine whether one of them can be pruned – i.e., whether there's a query on which its 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. If the models are not distinguishable, it doesn't make sense to try to prune one of them under the given query class. Formally,

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

In determining prunability, we need to 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,

(a) $\mathcal{M}^{\mathcal{A}}$'s $pick(?x)$ action (unknown to $\mathcal{H}$)

`(handempty),`	$\rightarrow$	`(in-hand(?x)),`
`(on-floor(?x))`		`(¬(handempty)),`
		`(¬(on-floor(?x)))`

(b) $\mathcal{M}_1$'s $pick(?x)$ action

`(handempty),`	$\rightarrow$	`(¬(handempty)),`
`(on-floor(?x))`		`(¬(on-floor(?x)))`

(c) $\mathcal{M}_2$'s $pick(?x)$ action

`(on-floor(?x))`	$\rightarrow$	`(¬(on-floor(?x)))`

(d) $\mathcal{M}_3$'s $pick(?x)$ action

`(on-floor(?x))`	$\rightarrow$	`()`

Figure 3: $pick$ actions of the agent model $\mathcal{M}^{\mathcal{A}}$ and three abstracted models $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$. Here $X \rightarrow Y$ means that X is the precondition of an action and Y is the effect.

i.e. $\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(\mathcal{M}')$. For plan outcome queries, consider that $\mathcal{Q}_{PO}(\mathcal{M}^{\mathcal{A}}) = \langle \ell, \langle p_1, \ldots, p_k \rangle \rangle$ and $\mathcal{Q}_{PO}(\mathcal{M}') = \langle \ell', \langle p'_1, \ldots, p'_j \rangle \rangle$. Now we can say that $\mathcal{Q}_{PO}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}_{PO}(\mathcal{M}')$ iff $(\ell = \ell')$, $j \leq k$ and $\forall i \in \{1, \ldots, j\} \bigwedge_i p_i = \bigwedge_i p'_i$.

Definition 3. Given an agent-interrogation task $\langle \mathcal{M}^{\mathcal{A}}, \mathbb{Q} \rangle$, two models M_i and M_j are *prunable* denoted as $\mathcal{M}_i \langle\rangle \mathcal{M}_j$, iff $\exists \mathcal{Q} \in \mathbb{Q} : M_i \rceil\lfloor^{\mathcal{Q}} M_j \wedge (\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(M_i) \wedge \mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \not\models \mathcal{Q}(M_j)) \vee (Q(\mathcal{M}^{\mathcal{A}}) \not\models \mathcal{Q}(M_i) \wedge \mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(M_j))$.

2.1 Solving the Interrogation Task

Our approach iteratively generate 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 the chemistry lab robot discussed in the Introduction. Assume that $\mathcal{H}$ is considering two abstract models $\mathcal{M}_1$ and $\mathcal{M}_2$ having only the predicates $handempty$, $on\text{-}floor(?x)$ and the agent's model is $\mathcal{M}^{\mathcal{A}}$ (Fig. 3). $\mathcal{H}$ can ask the agent what will happen if $\mathcal{A}$ picks up bottle b2 after bottle b1. The agent would respond that it could execute the plan only till length 1, and the state at the time of this failure was $(on\text{-}floor(b2)) \wedge (in\text{-}hand(b1)) \wedge (\neg(handempty)) \wedge (\neg(on\text{-}floor(b1)))$.

Algorithm 1 shows our overall algorithm for interrogating autonomous agents. It takes the agent $\mathcal{A}$, the set of propositions $\mathbb{P}$, and set of all actions $\mathbb{A}$ 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 1) representing that we are starting at the most abstract node in model lattice.

In each iteration of the main loop (line 2), 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 n given by some input ordering of Γ. The correctness of the algorithm does not depend on this ordering. We then generate all the new sets of models at the current node represented by the new_models (line 3). We also initialize an empty set at each lattice node to store the pruned models (line 4).

Algorithm 1: Agent Interrogation Algorithm

Input: $\mathcal{A}, \mathbb{A}, \mathbb{P}$
Output: poss_models
1 poss_models $= \{\{\emptyset\}\}$;
2 **for** γ *in some input pal ordering* Γ **do**
3 new_models $\leftarrow$ poss_models $\times\{\gamma^+, \gamma^-, \gamma^\emptyset\}$;
4 pruned_models$= \{\emptyset\}$;
5 **for** *each* $\mathcal{M}^{abs}$ *in poss_models* **do**
6 $\{\mathcal{M}^+_\gamma, \mathcal{M}^-_\gamma, \mathcal{M}^\emptyset_\gamma\} =$ $\{\mathcal{M}^{abs} \cup \gamma^+, \mathcal{M}^{abs} \cup \gamma^-, \mathcal{M}^{abs} \cup \gamma^\emptyset\}$;
7 **for** *each pair* $\{\mathcal{M}_i, \mathcal{M}_j\}$ *in* $\{\mathcal{M}^+_\gamma, \mathcal{M}^-_\gamma, \mathcal{M}^\emptyset_\gamma\}$ **do**
8 $\mathcal{Q} \leftarrow$ generate_query$(\mathcal{M}_i, \mathcal{M}_j)$;
9 $\mathcal{M}_{prune} \leftarrow$filter_models$(\mathcal{Q}, \mathcal{M}^{\mathcal{A}}, \mathcal{M}_i, \mathcal{M}_j)$;
10 pruned_models$\leftarrow$ pruned_models$\cup\mathcal{M}_{prune}$
11 **end**
12 **end**
13 **if** *pruned_models is* $\emptyset$ **then**
14 update_pal_ ordering(Γ);
15 continue;
16 **end**
17 new_models $\leftarrow$ new_models $\setminus$ pruned_models;
18 poss_models$\leftarrow$ poss_models $\cup$ new_models ;
19 **end**

The inner loop (line 5) iterates over the set of all possible models $poss\_models$. Each abstract model represented by $\mathcal{M}^{abs}$ is then refined with the pal tuple γ giving three different models and form pairs from these models and iterate over these pairs (line 6 and 7). Here $\mathcal{M}^m_\gamma$ represents the abstract models equivalent to $\mathcal{M}^{abs} \cup \{\gamma^m\}$, where $m \in \{+, -, \emptyset\}$.

For each pair, we generate a query $\mathcal{Q}$ using *generate_query()* that can distinguish between the models in that pair (line 8). 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 9). Then we update the estimated set of possible models represented by $poss\_models$ (line 17 and 18).

If we are unable to prune any models at a node (line 13), we update the order in which pal tuples are considered for refinement (line 14). 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. This algorithm would require $O(|\mathbb{A}| \times |\mathbb{P}|)$ queries. However, our empirical studies show that we never generate so many queries. Section 2.2 describes the *generate_query()* (line 8) component of the algorithm, Section 2.3 descibes the *filter_models()* (line 9) component, and Section 2.4 describes the *update_pal_ordering()* component (line 14).

2.2 Query Generation

The query generation process corresponds to *generate_query()* module in algorithm 1 which takes 2 models $\mathcal{M}_i$ and $\mathcal{M}_j$ as input and generates a query $\mathcal{Q}$ that can distinguish them, and if possible, satisfy prunability condition too.

Plan outcome queries can distinguish between models differing in either preconditions or effects of some action. We reduce the problem of creating plan outcome queries to a planning problem. The idea is to maintain a separate copy $\mathcal{P}^{\mathcal{M}_i}$ and $\mathcal{P}^{\mathcal{M}_j}$ of all the propositions $\mathcal{P}$, and formulate each precondition and effect of an action as a formula of predicates in both the copies of the propositions.

Let the planning problem $P_{PO} = \langle \mathcal{M}^{PO}, s_\mathcal{I}, s_\mathcal{G} \rangle$, where $\mathcal{M}^{PO}$ is a model with propositions $\mathcal{P}^{PO} = \mathcal{P}^{\mathcal{M}_i} \wedge \mathcal{P}^{\mathcal{M}_j} \wedge p_\gamma$, and actions $\mathbb{A}$ where for each action $a \in \mathbb{A}$, $pre(a) = pre(a^{\mathcal{M}_i}) \vee pre(a^{\mathcal{M}_j})$ and $eff(a) = (when(pre(a^{\mathcal{M}_i}) \wedge pre(a^{\mathcal{M}_j}))(eff(a^{\mathcal{M}_i}) \wedge eff(a^{\mathcal{M}_j}))\ (when((pre(a^{\mathcal{M}_i}) \wedge \neg pre(a^{\mathcal{M}_j})) \vee (\neg pre(a^{\mathcal{M}_i}) \wedge pre(a^{\mathcal{M}_j})))\ (p_\gamma))$, $s_\mathcal{I} = s_\mathcal{I}^{\mathcal{M}_i} \wedge s_\mathcal{I}^{\mathcal{M}_j}$ 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 $s_\mathcal{G}$ is the goal state and it is expressed as p_γ.

With this formulation whenever we have at least one action in both the models which has different effects in both of them, the goal will be reached. For example, consider the models $\mathcal{M}^{\mathcal{A}}$ and $\mathcal{M}_1$ mentioned in Fig. 3. On applying the $pick(b1)$ action from the state where the action can be applied in both the models, one of them will lead to $in\text{-}hand(b1)$ being true and the other will not. Hence starting with an initial state $s_\mathcal{I} = on\text{-}floor(b1) \wedge handempty$, the plan to reach the goal will be $pick(b1)$.

Also, whenever we have an action a which cannot be applied in the same state s_d in both the models, the planner will generate a plan to take the agent from the initial state to state s_d, and append action a to that plan. This new plan will be the solution to the planning problem P_{PO}. For example, consider the models $\mathcal{M}_1$ and $\mathcal{M}_2$ mentioned in Fig. 3. In a state where $on\text{-}floor(b1)$ is true and $handempty$ is false, we can apply $pick(b1)$ in $\mathcal{M}_2$ but not in $\mathcal{M}_1$. Hence for an initial state $s_\mathcal{I} = on\text{-}floor(b1) \wedge \neg handempty$, the plan to reach the goal will be $pick(b1)$. The following theorem formalizes these notions.

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). $\mathcal{Q}_{PO}$ comprises of an initial state $s_\mathcal{I}$ and plan π. The initial state $s_\mathcal{I}$ in $\mathcal{Q}_{PO}$ and P_{PO} is same. Starting with this initial state, an action becomes a part of the plan π only when it can be applied in any one or both of the models $\mathcal{M}_i$ and $\mathcal{M}_j$. So two cases arise here, if the action can be executed in both the models, the effect of both the actions is applied to the state and next action is searched. Otherwise if the action is applicable only in one of the models, but not the other, the effect of the action is a dummy proposition p_γ which is also the goal. So as soon an action is found that is possible in one of the models but not the other, or if it gives different resulting states in both the models, the resulting plan becomes the plan needed by query $\mathcal{Q}_{PO}$. Hence if the planning problem P_{PO} gives a solution plan π, then there exists a query $\mathcal{Q}_{PO}$ that consists of $s_\mathcal{I}$ and π as input.

Also, as described previously, whenever there exists a distinguishing plan-outcome query, the starting state $s_\mathcal{I}$ is part of $\mathcal{Q}_{PO}$, and the way we generate the P_{PO} problem ensures we will get a plan π as the solution. □

2.3 Filtering Possible Models

This section describes the *filter_models()* module in algorithm 1 which takes as input the agent model $\mathcal{M}^{\mathcal{A}}$, the two abstract models being compared $\mathcal{M}_i$ and $\mathcal{M}_j$, 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 $\mathcal{M}^{\mathcal{A}}$.

Firstly, the algorithm *asks the query* $\mathcal{Q}$ to both the models $\mathcal{M}_i$ and $\mathcal{M}_j$ 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. 3, 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.

If the models are prunable, let the model not consistent with the agent be $\mathcal{M}'$ where $\mathcal{M}' \in \{\mathcal{M}_i, \mathcal{M}_j\}$. Now recall that a model is a set of palm tuples. As shown in Fig. 2, based on response to a query, if a model is found to be inconsistent for the first time at a node n in the lattice, with an incoming edge of label γ, any model with same mode of γ as $\mathcal{M}'$ will also be inconsistent. This is because a palm tuple uniquely identifies the mode in which a predicate will appear in an action's location which can be precondition or effect. And since this tuple is inconsistent with the agent, any model containing this will also contain the same mode of predicate in that action's precondition or effect. This idea paves way for the concept of partitions which is discussed below.

Given lattice nodes n_i and n_j, the edge $n_j \rightarrow n_i$ labeled γ, and the set Λ of palm tuples present at the parent node n_j, a *partition* of node n_i is the set of disjoint subsets $\Lambda \cup \{\gamma^+\}$, $\Lambda \cup \{\gamma^-\}$, and $\Lambda \cup \{\gamma^{\emptyset}\}$. So depending on the model $\mathcal{M}'$ which is inconsistent with agent model $\mathcal{M}^{\mathcal{A}}$, we can prune out the whole partition containing $\mathcal{M}'$. This partition is returned by *filter_models()* module as $\mathcal{M}_{prune}$.

2.4 Updating pal ordering Γ

Models may not be prunable if the query is not executable by $\mathcal{A}$ and none of the model's query responses are consistent with that of the agent. For eg, consider two abstract models $\mathcal{M}_2$ and $\mathcal{M}_3$ being considered by the human interrogator $\mathcal{H}$ (Fig. 3). At this level of abstraction, $\mathcal{H}$ does not have knowledge of predicate $handempty$, hence it will generate a plan outcome query with initial state $on\text{-}floor(b1)$ and plan $pick(b1)$ to distinguish between $\mathcal{M}_2$ and $\mathcal{M}_3$. But this cannot be executed by agent $\mathcal{A}$ as the precondition $handempty$ is not satisfied. In such cases, we cannot discard any partitions. Hence if no prunable query is possible, i.e. the palm tuple set Λ being considered is last in $poss\_models$, we update the pal ordering. Recall that in response to the plan outcome query we get the failed action $a_{Fail} = \pi[\ell]$ and the final state $s_{\mathcal{F}}$. Let us assume that the query was executable on $\mathcal{M}_i$, but not on $\mathcal{M}_j$. Now assuming $\mathcal{M}_i$ is an abstracted version of $\mathcal{M}^{\mathcal{A}}$, let the state it reaches after executing first ℓ steps of the plan be $\overline{s}_{\mathcal{F}}$. Now we can infer that one of the literal present in $s_{\mathcal{F}} \setminus \overline{s}_{\mathcal{F}}$ (represented as $\{l_1, \ldots, l_k\}$) is causing the action a_{Fail} to fail. We now generate a new query with $s_{\mathcal{I}} = \overline{s}_{\mathcal{F}}$ and keeping a subset of $\{\neg l_1 \ldots \neg l_k\}$. With this $s_{\mathcal{I}}$ as initial state, the agent $\mathcal{A}$ should be able to execute the plan $\pi = a_{Fail}$. In next step we change the initial state $s_{\mathcal{I}}$ to $\overline{s}_{\mathcal{F}} \wedge l_k$ and remove l_k from the subset we found earlier. If $\mathcal{A}$ still executes $\pi = a_{Fail}$, then l_k was not the literal responsible for the failure of a_{Fail} and we change $s_{\mathcal{I}}$ to $\overline{s}_{\mathcal{F}} \wedge l_k \wedge l_{k-1}$, otherwise we can infer that l_k was indeed one of the literals responsible for failure of a_{Fail}, and we change $s_{\mathcal{I}}$ to $\overline{s}_{\mathcal{F}} \wedge \neg l_k \wedge l_{k-1}$. We do this k times to determine the literals responsible for failure of action a_{Fail}. For each of such literals causing the failure, we get their correct palm tuples. For eg. if we inferred that $on\text{-}table(?x)$ was not present in the state and hence was causing the action $pick(?x)$ to fail, we get the correct palm tuple as $\langle on\text{-}table(?x), pick(?x), precondition, + \rangle$. We need not refine in terms of the corresponding pal tuple $\langle on\text{-}table(?x), pick(?x), precondition \rangle$ in future, so we remove it from the pal ordering Γ.

2.5 Correctness of Agent Interrogation Algorithm

In this section we prove that the set of estimated models returned by the agent interrogation algorithm are correct. A good starting point will be to verify the correctness of the queries. In the following theorem we show that if we pick the models from different partitions to generate the query (Step 7 of the algorithm 1), then we will always get distinguishing queries. We break this into 2 parts, first we consider the case where the location l in the pal tuple is precondition, then we discuss about the pal tuples with effect as the location.

Theorem 2. For the refinement in terms of *pal* tuple $\gamma = \langle p, a, l = precondition \rangle$ of two models $\mathcal{M}_i$ and $\mathcal{M}_j$, if $\mathcal{M}_i$ and $\mathcal{M}_j$ are not distinguishable $\mathcal{M}_i \not\downarrow \mathcal{M}_j$, then their refinements when adding γ will be distinguishable only if the refinements belong to different partitions i.e., $(\mathcal{M}_i \cup \gamma^{m_1}) \downarrow (\mathcal{M}_j \cup \gamma^{m_2})$ if $m_1 \neq m_2$ and $(\mathcal{M}_i \cup \gamma^{m_1}) \not\downarrow (\mathcal{M}_j \cup \gamma^{m_2})$ if $m_1 = m_2$, where $m_1, m_2 \in \{+, -, \emptyset\}$.

Proof (Sketch). Let us consider 2 different cases;
First case when $m_1 = m_2$, i.e., both $(\mathcal{M}_i \cup \gamma^{m_1})$ and $(\mathcal{M}_j \cup \gamma^{m_2})$ are in the same partition. Let us assume that a query $\mathcal{Q}$ with plan $\pi_{\mathcal{Q}}$ exists that can distinguish between the refined models $(\mathcal{M}_i \cup \gamma^{m_1})$ and $(\mathcal{M}_j \cup \gamma^{m_2})$. Now since before refinement $\mathcal{M}_i \not\downarrow \mathcal{M}_j$ and the only change we have made in the models is in action a, it should be part of the plan π. More specifically it should be the last action in the plan as after applying this action, both models will result in different states. But the change being made in both $\mathcal{M}_i$ and $\mathcal{M}_j$ is same, hence this action a cannot distinguish between the two models, so we reach a contradiction with our initial assumption, hence $(\mathcal{M}_i \cup \gamma^{m_1}) \not\downarrow (\mathcal{M}_j \cup \gamma^{m_2})$ if $m_1 = m_2$.

For the second case when $m_1 \neq m_2$ i.e., $(\mathcal{M}_i \cup \gamma^{m_1})$ and $(\mathcal{M}_j \cup \gamma^{m_2})$ are in different partitions, we can safely assume that we have at least one action a which is part of γ that can distinguish between the two models. When $l = precondition$, we can generate a state $s_{\mathcal{I}}$ where this action is applicable in one model but not in another. More specifically, if $\{m_1, m_2\} = \{+, -\}$, we can generate an initial state $s_{\mathcal{I}}$ with literal p or $\neg p$, in both the cases, only one of the models will be able to execute the plan π. If $\{m_1, m_2\} = \{+, \emptyset\}$, we can generate an initial state $s_{\mathcal{I}}$ with literal $\neg p$, and hence only the model with $m = \phi$ will

(a) $\mathcal{M}_1$'s *put-on-shelf*$(?x, ?y)$ action
$\lambda^{\emptyset} = \langle$*same-label*$(?x, ?y)$, *put-on-shelf*$(?x, ?y)$, *eff*,$\emptyset\rangle$

```
(in-hand(?x)),          →   (not(in-hand(?x)),
(same-label(?x,?y))         (handempty),
                            (on-shelf(?x,?y))
```

(b) $\mathcal{M}_2$'s *put-on-shelf*$(?x, ?y)$ action
$\lambda^{+} = \langle$*same-label*$(?x, ?y)$, *put-on-shelf*$(?x, ?y)$, *eff*,$+\rangle$

```
(in-hand(?x)),          →   (not(in-hand(?x))),
(same-label(?x,?y))         (handempty),
                            (on-shelf(?x,?y)),
                            (same-label(?x,?y))
```

Figure 4: Two model variants that are functionally equivalent

be able to execute the plan π to completion. And finally if $\{m_1, m_2\} = \{-, \emptyset\}$, we can generate an initial state $s_{\mathcal{I}}$ with literal p, and hence only the model with $m = \emptyset$ will be able to execute the plan π to completion. □

We cannot use the exact same argument when refinement is done for the effects because we can have models with different effects that are functionally equivalent. An intuitive example of this will be a pair of models where in one of the model the same literal appears in both the preconditions and effects, whereas in the other model the same literal appears only in the precondition.

For example, consider the chemistry lab robot that cross-checks for the labels of the bottles when placing the chemical bottles in correct shelves. Assume we have two models $\mathcal{M}_1$ and $\mathcal{M}_2$ with the two versions of the action *put-on-shelf*$(?x, ?b)$ as shown in Fig. 4. In this case, we cannot generate an initial state $s_{\mathcal{I}}$ that can lead to plan that can distinguish between the models with these two different refinements. In simpler terms, the planning problem P_{PO} discussed in Theorem 1 will not give a solution in this case.

Theorem 3. For the refinement in terms of *pal* tuple $\gamma = \langle p, a, l = \textit{effect}, m\rangle$ of two models $\mathcal{M}_i$ and $\mathcal{M}_j$, if $\mathcal{M}_i$ and $\mathcal{M}_j$ are not distinguishable $\mathcal{M}_i \nparallel \mathcal{M}_j$, then their refinements when adding palm tuple λ will be distinguishable only if the refinements belong to separate partitions except when one of the partition corresponds to tuple $\gamma^{\emptyset}$ and $\gamma' = \langle p, a, l = precondition, m = \{+, -\}\rangle \in \mathcal{M}_i \cap \mathcal{M}_j$.

Proof (Sketch). Let us consider 2 different cases; First case when $m_1 = m_2$, i.e., both $(\mathcal{M}_i \cup \lambda^{m_1})$ and $(\mathcal{M}_j \cup \lambda^{m_2})$ are in the same partition. Let us assume that a query $\mathcal{Q}$ with plan $\pi_{\mathcal{Q}}$ exists that can distinguish between the refined models $(\mathcal{M}_i \cup \gamma^{m_1})$ and $(\mathcal{M}_j \cup \gamma^{m_2})$. Now since before refinement $\mathcal{M}_i \nparallel \mathcal{M}_j$ and the only change we have made in the models is in action a, it should be part of the plan π. More specifically it should be the last action in the plan as this action cannot be applied in one of the models. But the change being made in both $\mathcal{M}_i$ and $\mathcal{M}_j$ is same, hence this action a cannot distinguish between the two models, so we reach a contradiction with our initial assumption, hence $(\mathcal{M}_i \cup \gamma^{m_1}) \nparallel (\mathcal{M}_j \cup \gamma^{m_2})$ if $m_1 = m_2$.

For the second case when $m_1 \neq m_2$ i.e., $(\mathcal{M}_i \cup \gamma^{m_1})$ and $(\mathcal{M}_j \cup \gamma^{m_2})$ are in different partitions, if we consider only the cases where the following condition holds for two palm tuple variants being considered: $\{\langle p, a, l = precondition, m_1\rangle \in \mathcal{M}_i \cap \mathcal{M}_j\} \wedge \{\langle p, a, l = \textit{effect}, m_2\rangle \in \mathcal{M}_i \cap \mathcal{M}_j\} \Rightarrow m_1 \neq m_2$; then we can safely assume that we have at least one action a which is part of γ that can distinguish between the two models. When $l =$ *effect*, we can generate a state $s_{\mathcal{I}}$ where this action is applicable, then according to Theorem 1, we can get always get a plan π that will distinguish between the two models. □

The theorems given above prove that the way we pick models to generate a query gives us distinguishing queries in almost all cases. And for the cases it does not generate a distinguishing query, we do not prune any model. We now prove that the algorithm prunes away a model only when it is inconsistent with the responses given by the agent. Here we are assuming that the agent has deterministic actions, so we can safely infer that an inconsistent answer will always be inconsistent.

Theorem 4. If we prune away an abstract model $\mathcal{M}^{abs}$, then no possible concretization of $\mathcal{M}^{abs}$ will result into a model consistent with the agent model $\mathcal{M}^{\mathcal{A}}$.

Proof (Sketch). At each node in the lattice, we always prune away some of the models, this pruning happens in one of the two possible ways:

First, when the distinguishing query between two models (which are variants of each other) is executed successfully by the agent $\mathcal{A}$, we discard a model that is not consistent with the agent. We discard a model only when there is a proposition in the abstracted version of the final state of the agent that is not present in the final state of the model. So for any concretized version of the model, the number of states reachable will never increase, so any concretized version of that model cannot make those propositions (which were false when we initially discarded) true, so it is never possible for a concretized version of a discarded model to be always consistent with the agent model.

Second case is easier to deal here. Whenever we are not able to generate a plan that is fully executable on the agent $\mathcal{A}$, we get new refinement tuples $\langle p_{Fail}, a_{Fail}, l = precondition, m_{Fail}\rangle$ (inferred from $update\_pal\_ordering()$ module). This is akin to the agent telling us indirectly that this is the correct form of this palm tuple, hence we can discard other variants of this tuple with full surety. □

With the guarantee that we are not pruning away any correct possible model, we now move on to prove that the agent interrogation algorithm will terminate, hence giving a solution always.

Theorem 5. The Agent Interrogation Algorithm mentioned in algorithm 1 will always terminate.

Proof (Sketch). At each step of the algorithm, when we consider a refinement in terms of γ tuples, we are left with one or more variant of the γ tuple. This ensures that we never have to refine models more than once at a single level in the lattice. We are assuming level in our subset lattice, to be the number of refined γ tuples. Since we refine at least one γ tuple in every iteration of the algorithm, the algorithm is bound to terminate as the number of γ tuples is finite for a

Domain	$\lvert\mathbb{P}^*\rvert$	$\lvert\mathbb{A}\rvert$	$\lvert\overline{\mathcal{M}}\rvert$	$\lvert\overline{\mathcal{Q}}\rvert$	Algorithm 1	
					$\lvert\hat{\mathcal{Q}}\rvert$	Time/$\mathcal{Q}$ (sec)
gripper	5	3	9^{15}	$15 * 2^5$	37	0.14
blocks*	9	4	9^{36}	$36 * 2^9$	92	1.73
elevator	10	4	9^{40}	$40 * 2^{10}$	109	5.91
logistics	11	6	9^{66}	$66 * 2^{11}$	98	11.62
parking	18	4	9^{72}	$72 * 2^{18}$	173	12.01
satellite	17	5	9^{85}	$85 * 2^{17}$	127	19.53
stacks$	10	12	9^{120}	$120 * 2^{10}$	203	11.28

Table 1: Comparison of the number of queries and average time per query in our approach vs a naive baseline. $|\hat{\mathcal{Q}}|$ denotes the number of queries used in our approach, as opposed to the number of queries $|\overline{\mathcal{Q}}|$ that would be required in a naive solution having $|\overline{\mathcal{M}}|$ possible models to start with (see Sec. 3. Times shown are averages of time taken per query across 10 runs of the agent interrogation algorithm.
Full name of IPC domain is *blocks-world, $openstacks.

finite number of propositions and actions under consideration. □

In the last theorem we proved that agent interrogation algorithm will always give us a solution, we now prove that the solution given by the algorithm is correct.

Theorem 6. As part of its solution, Agent Interrogation algorithm always gives a set of models, each of which are functionally equivalent to agent's model $\mathcal{M}^{\mathcal{A}}$.

Proof (Sketch). As a solution to the agent interrogation algorithm, we might get multiple possible models each of which are functionally equivalent. Theorem 5 sets up an invariant for the agent interrogation algorithm that is the number of refined palm tuples increase as the number of iterations of the algorithm increase. This property when combined with theorem 4 ensures that at each step only the incorrect models are discarded. So the models leftover at the most concretized node after all the palm tuples are refined are going to be the set of models which the algorithm could not discard, hence all of these models are guaranteed to be functionally equivalent to the agent model. □

3 Empirical Evaluation

The approach developed in this paper uses very sparse but selective information from the agent to infer its mode. Existing approaches for model learning (see Sec. 1) cannot work with such information (outcomes of plans but not their intermediate states). However, we can consider a naive method for solving the problem as a baseline: all possible models can be generated and then their answers to queries are compared to agent's answers. This method is guaranteed to find the solution but the complexity of this approach is exponential in the number of predicates.

To test our approach, in each run we created an agent (unknown to Alg. 1) with one of the 7 IPC domain models. This agent supports plan outcome queries. We then evaluated the performance of Alg. 1 in estimating that agent's model using 10 different problems from that domain. We generate initial

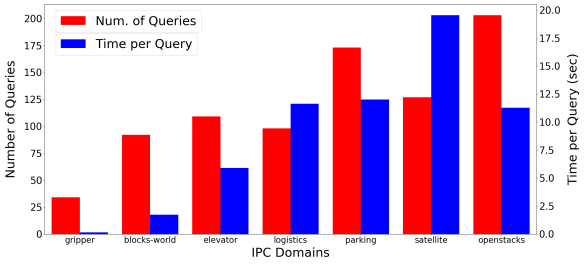

Figure 5: Bar chart showing the number of queries and time per query for the seven IPC domains.

states for queries using an increasing number of objects until a distinguishing query is found. In all our experiments, such a query was found using at most 7-8 objects. This number is found to correlate with the maximum arity of the predicates in the domain. Hence the number of queries is not affected by the number of objects as the approach finds the minimum number of objects needed to distinguish between the abstracted models and uses it, and providing more objects does not change the behaviour of the algorithm. This was further validated from the test runs as the value of $|\hat{\mathcal{Q}}|$ for a domain did not change across the 10 problems they were tested on. Also, the number of expected models was always one as the corner case shown in Fig. 4 was not present in any of the 7 domains. All the experiments were run on a 4.9 GHz Ubuntu machine with 64 GB RAM.

Table 1 and Fig. 5 illustrate our results. Table 1 shows that the number of queries asked ($|\hat{\mathcal{Q}}|$) in our approach is much smaller than that needed for the naive method ($|\overline{\mathcal{Q}}|$).

Also, there is no definite pattern in the number of queries asked as the order in which queries were asked (depending of ordering of γ tuples) was random. So a better query asked earlier in the interrogation process can lead to a smaller number of queries asked.

4 Conclusion

We have presented a novel approach for estimating the model of an autonomous agent by interrogating it. In this paper we showed that the number of queries required to estimate the model is dependent only on the number of actions and predicates, and is independent of the size of the environment.

Extending this approach to more general types of agents and environments featuring partial observability and/or nondeterminism is a promising direction for future work.

Although our interface is a set of plans represented as logical statements, other works have explored using natural language as a way to provide plans as input (Lindsay et al. 2017). In future, this can be used to extend our work thereby making the communication more realistic and close to how a human interrogator might actually interact with an autonomous agent.

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

Aineto, D.; Jiménez, S.; Onaindia, E.; and Ramírez, M. 2019. Model recognition as planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, 13–21.

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

Amir, E., and Russell, S. 2003. Logical filtering. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-03*, volume 3, 75–82.

Bäckström, C., and Jonsson, P. 2013. Bridging the gap between refinement and heuristics in abstraction. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, 2261–2267. AAAI Press.

Bryce, D.; Benton, J.; and Boldt, M. W. 2016. Maintaining evolving domain models. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 3053–3059. AAAI Press.

Camacho, A., and McIlraith, S. A. 2019. Learning interpretable models expressed in linear temporal logic. In *International Conference on Automated Planning and Scheduling*. ICAPS.

Cresswell, S., and Gregory, P. 2011. Generalised domain model acquisition from action traces. In *International Conference on Automated Planning and Scheduling*.

Cresswell, S.; McCluskey, T.; and West, M. 2009. Acquisition of object-centred domain models from planning examples. In *International Conference on Automated Planning and Scheduling*.

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.

Fox, M., and Long, D. 2003. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *JAIR* 20(1):61–124.

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

Helmert, M.; Haslum, P.; Hoffmann, J.; et al. 2017. Flexible abstraction heuristics for optimal sequential planning. In *International Conference on Automated Planning and Scheduling*.

Helmert, M. 2009. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence* 173(5):503 – 535.

Khardon, R., and Roth, D. 1996. Reasoning with models. *Artif. Intell.* 87(1-2):187–213.

Kučera, J., and Barták, R. 2018. Louga: Learning planning operators using genetic algorithms. In Yoshida, K., and Lee, M., eds., *Knowledge Management and Acquisition for Intelligent Systems*, 124–138. Cham: Springer International Publishing.

Lindsay, A.; Read, J.; Ferreira, J.; Hayton, T.; Porteous, J.; and Gregory, P. 2017. Framer: Planning models from natural language action descriptions. In *International Conference on Automated Planning and Scheduling*.

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

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

Srivastava, S.; Russell, S.; and Pinto, A. 2016. Metaphysics of planning domain descriptions. In *AAAI Conference on Artificial Intelligence*.

Stern, R., and Juba, B. 2017. Efficient, safe, and probably approximately complete learning of action models. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 4405–4411.

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