Learning Operators with Ignore Effects for Bilevel Planning in Continuous Domains

Nishanth Kumar*, Willie McClinton*, Rohan Chitnis, Tom Silver, Tomás Lozano-Pérez, Leslie Pack Kaelbling

MIT Computer Science and Artificial Intelligence Laboratory
{njk, wbm3, ronuchit, tslvr, tlp, lpk}@csail.mit.edu

Abstract

Bilevel planning, in which a high-level search over an abstraction of an environment is used to guide low-level decision making, is an effective approach to solving long-horizon tasks in continuous state and action spaces. Recent work has shown that action abstractions that enable such bilevel planning can be learned in the form of symbolic operators and neural samplers given symbolic predicates and demonstrations that achieve known goals. In this work, we show that existing approaches fall short in environments where actions tend to cause a large number of predicates to change. To address this issue, we propose to learn operators with ignore effects. The key idea motivating our approach is that modeling every observed change in the predicates is unnecessary: the only changes that need be modeled are those that are necessary for high-level search to achieve the specified goal. Experimentally, we show that our approach is able to learn operators with ignore effects across six hybrid robotic domains that enable an agent to solve novel variations of a task, with different initial states, goals, and numbers of objects, significantly more efficiently than several baselines.

1 Introduction

Solving long-horizon tasks in domains that have continuous state and action spaces is challenging, even when the transition function is deterministic and known. One effective solution is to construct high-level state and action abstractions, plan to solve the task in this abstract space that is easy to search, then use the resulting plan to guide search through the low-level continuous space. Typically, the state and action abstractions leveraged by this bilevel planning approach are hand-specified. In this work, we address the problem of learning action abstractions from data.

An abstract domain model consists of a set of predicates that induce a state abstraction, operator descriptions in terms of those predicates that describe a (partial) transition model, and samplers that enable the search for realizations of abstract actions in terms of primitive actions. A typical approach to learning an abstract domain model (e.g., Silver et al. (2022)) is to search in an outer loop over possible predicates, and then in an inner loop to find operator descriptions and samplers given a predicate set. In this paper, we focus on the problem of learning operator descriptions given a set of predicates (classifiers on segments of the low-level state, such as InGripper), an accurate low-level transition model, and a set of parameterized controllers (such as Pick(x, y, z)) that serve as primitive actions. The learning algorithm operates on a very small number of demonstration trajectories, and by learning an abstract model, generalizes aggressively to a highly variable set of problem domain sizes, initial states, and goals. It produces both symbolic abstract operator descriptions and samplers that select appropriate continuous parameters for the controllers associated with the operators.

A natural objective for the problem of finding good domain abstractions is prediction accuracy (i.e., minimizing prediction error) (Silver et al. 2021a, 2022): many metrics, including bisimulation (Dean and Givan 1997), ask that the high-level model accurately predict the next abstract state given the current abstract state and action. This objective would be appropriate if we were using the high-level model to make predictions, but in fact we are using it as search-guidance for an accurate low-level model. So instead, our objective is to find an abstract model that maximally improves the performance of the search algorithm, given the available data. The difference between these objectives is stark in real-world domains with many objects and predicates, and in which each action taken by the agent changes a large number of propositions. To make highly accurate predictions about all of these state changes might require a very fine-grained model with many complex operators. Such a model would require a lot of data to learn reliably and would be very slow to plan with.

As a simple example of this phenomenon, consider an environment (illustrated in Figure 1) where a robot operates a crane that can magnetize itself to pick up all screws within a small distance of its gripper, although its goal is only ever to pick up one particular golden screw. A Pickable(scr) predicate, which is true if the distance between the crane face and a particular screw is small enough, seems to be a sensible precondition for engaging the magnet in order to pick up a screw. However, as a result of any given motion of the robot, a varying number of instances of the Pickable predicate will become true or false, depending on the arrangement of screws. Modeling this phenomenon accurately requires a complicated set of operators (e.g., see Figure 1, bottom left) that is highly overfit to the training data and...
will be incorrect, slow, or both, on new problem instances.

In this work, we take seriously the objective of generating useful high-level plans. Operators need only model changes in predicates that are necessary for high-level search to find a good operator sequence that can be refined through low-level planning. More formally, we say that a symbolic effect is necessary if it appears in the goal, or (recursively) if it is a precondition of an operator with a necessary effect. For example, an operator for moving in the ‘screws’ domain has only the necessary effect of making the golden screw pickable, even if it might also make many others pickable.

In order to model only such necessary effects, we change the typical STRIPS formalism to include universally quantified “ignore effects”, which allow us to declare, for example, that we will not commit to the truth value of instances of Pickable, except for those that are explicitly declared in the operator’s add effects (shown in Figure 1). This change in interpretation has almost no effect on the planning algorithm, but gives substantial freedom to the operator-learning method, which can be used to avoid overfitting and to maintain efficiency of the resulting planner.

Our main contributions are (1) a generalization of STRIPS operators to include ignore effects and (2) a novel algorithm for efficiently learning these operators from a small number of examples. We test this method through experiments on a wide range of robotic planning problems with continuous state and action spaces, and demonstrate that its ability to learn to solve complex problems and generalize substantially from a small number of examples is superior to several baseline operator learning methods, as well as imitation learning methods that learn a policy directly.

2 Background

2.1 Problem Setting

We assume that our planner will operate in the context of a system with the perceptual and memory ability to produce a low-level “scene” representation consisting of a set of objects, each of which is characterized by a vector of properties (such as pose, color, material, type). We are given predicates that classify objects based on these properties and a set of controllers that take a list of objects together with some continuous parameters and produce a low-level action sequence. We call this overall setting an environment.

In our setting, an environment is characterized by a tuple \((\Lambda, \mathcal{U}, f, \Psi, C)\). Here, \(\Lambda\) refers to a set of object types, where each \(\lambda \in \Lambda\) has a name (e.g., screw, robot) and a vector of \(\dim(\lambda)\) real-valued features (e.g., \((x, y, z, length, color, ...))\). The environment’s action space is denoted by \(\mathcal{U} \subseteq \mathbb{R}^m\). The transition function, which is a deterministic map from state, action pairs to a next state, is denoted by \(f : \mathcal{X} \times \mathcal{U} \rightarrow \mathcal{X}\). The environment’s set of predicates is denoted by \(\Psi\), where each predicate \(\psi \in \Psi\) has a name (e.g., Holding) and a tuple of types (e.g., (gripper, screw)). Finally, \(C\) denotes a finite set of controllers. Each
controller \(C((\lambda_1, \ldots, \lambda_n), \Theta) \in C\) has discrete typed variables \((\lambda_1, \ldots, \lambda_n)\), and a continuous real-valued vector of parameters \(\Theta\). For instance, a controller \texttt{Pick} for picking up a screw might have one variable of type \texttt{screw} and a \(\Theta\) that specifies a grasp transform.

Every environment is associated with a task distribution \(T\), where each \(T \in T\) is a tuple \((O, \varphi, \mu)\). Here, \(O\) denotes the set of objects in the particular task \(T\). Every object \(o \in O\) has a name (e.g., \texttt{screw}) and a type, denoted \texttt{type}(\(o\)) \(\in \Lambda\). Objects in turn give rise to the notion of state. We define a state \(x \in X\) to be an assignment of objects to feature vectors, that is, \(x(\text{\texttt{type}}(o)) \in \mathbb{R}^{\text{dim}(	ext{\texttt{type}}(o))}\) for \(o \in O\). Predicates induce a state abstraction: \(\text{\texttt{abstract}}(x)\) denotes the set of ground atoms that are known to be true in \(x\). A ground atom is a predicate combined with a mapping from its type tuple to objects (e.g., \texttt{Holding}\((?g, ?s)\)). By contrast, a lifted atom instead has a mapping to typed variables, which are placeholders for objects (e.g., \texttt{Holding}(?\texttt{g}, ?\texttt{s})). We use \(s \in \mathcal{S}\) to denote an abstract state, i.e., \(\text{\texttt{abstract}} : X \rightarrow \mathcal{S}\). Note that abstract is a many-to-one mapping, so many different states may map to the same abstract state. Finally, \(x_0 \in X\) denotes the task's initial state, while \(g\) denotes the task goal, which we assume is always a set of ground atoms. Note that a goal can be achieved by a set of abstract states: for example, there are many possible abstract states \(s\) where \(g = \{\text{InReceptacle(screw3)}\} \subseteq s\).

A solution to a task is a sequence of actions \(\pi = (u_1, \ldots, u_n)\) that achieve the goal. Specifically, \(g \subseteq \text{\texttt{abstract}}(x_n)\), and \(x_i = f(x_{i-1}, u_i)\) for \(1 \leq i \leq n\). While it is conceptually possible to plan over environment actions, it is much more feasible to plan over controllers that can be invoked to output low-level actions. Thus, we will henceforth take plans to be sequences of ground skills \(\phi = (\phi_1, \ldots, \phi_k)\) that similarly achieve the goal. A skill \(\phi\) refers to a controller combined with a mapping from its typed parameters to specific objects that match this type. Thus, a ground skill \(\phi\) is a skill combined with a specification of its continuous real-valued parameter vector \(\Theta\) and defines a low-level policy \(\phi : X \rightarrow \mathcal{U}\) along with a termination function. Ground skills return a sequence of low-level actions.

We consider a standard learning setting where a set of tasks \(\mathcal{T}_{\text{train}}\) drawn from \(T\) are available at training time, and held-out evaluation tasks drawn from \(T\) are used for evaluation. Each training task \(T\) is associated with a \(k\)-length demonstration \(\langle T, \omega, \Psi \rangle\), where \(\omega\) is a solution for \(T\), and where state sequence \(\Psi = (s_0, \ldots, s_k)\) with \(\text{\texttt{abstract}}(x_i) = s_i\) and \(x_i = f(x_{i-1}, \phi)\) for \(\phi_i \in \omega\) and \(1 \leq i \leq k\). Note that we do not assume the demonstrations are optimal, just that they are goal-achieving. The objective is to efficiently solve (i.e., with the shortest length plans) held-out tasks drawn from \(T\), using information learned from the provided demonstration set \(D\).

### 2.2 Operators, Samplers, and Bilevel Planning

Given demonstrations \(D\), what representations should be learned, and how should they be used for efficient and effective decision-making in evaluation tasks? Similar to Silver et al. (2021b, 2022), we consider a framework where symbolic operators and samplers are learned and then used for bilevel planning. A STRIPS operator is a tuple \(\omega = (\text{\texttt{VAR}}, P, E^+, E^-)\) where \(\text{\texttt{VAR}}\) is an ordered list of variables; \(P, E^+, E^-\) are positive preconditions, add effects, and delete effects respectively, each a set of lifted atoms over \(\Psi\) and \(\text{\texttt{VAR}}\); and \(C\) is a specific controller.

A ground STRIPS operator \(\omega = (\omega, \delta)\) is an operator \(\omega\) and a substitution \(\delta : \text{\texttt{VAR}} \rightarrow \mathcal{O}\) mapping its variables to objects. We use \(P, E^+, E^-\), and \(\phi\) to denote the ground preconditions, ground add effects, ground delete effects, and skill. A set of ground STRIPS operators induce an abstraction over the low-level environment formally defined as \((\Psi, S_\phi, \Omega, F, O)\). Here, \(S_\phi\) represents the space of high-level states, \(\Omega\) represents the space of high-level operators ground with objects \(O\), and partial function \(F : S_\Psi \times \Omega \rightarrow S_\Psi\) defines the high-level transition dynamics. The transition function \(F(s, \omega)\) is only defined if \(\omega\) is applicable in \(s\), that is \(P \subseteq s\). If defined, \(F(s, \omega) = F(s, \omega)\) defines the high-level transition dynamics. The transition function \(F(s, \omega)\) is only defined if \(\omega\) is applicable in \(s\), that is \(P \subseteq s\). If defined, \(F(s, \omega) = F(s, \omega)\) defines the high-level transition dynamics. The transition function \(F(s, \omega)\) is only defined if \(\omega\) is applicable in \(s\), that is \(P \subseteq s\). If defined, \(F(s, \omega) = F(s, \omega)\) defines the high-level transition dynamics.

Ground STRIPS operators cannot be executed directly in the environment. Rather, they must be refined into a ground skill. Since a ground operator \(\omega\) is associated with a skill \(\phi = C((o_1, \ldots, o_n), \Theta)\), refinement requires specifying values for the continuous vector of parameters \(\Theta\). For this, we employ samplers \(\Sigma\). A sampler outputs a distribution over \(\Theta\) given the low-level states of \((o_1, \ldots, o_n)\) as input. Formally, it is defined as \(\sigma : X \times O^N \rightarrow \Delta(\Theta)\), where \(\Delta(\Theta)\) is the space of distributions over \(\Theta\). Each operator \(\omega \in \Omega\) is associated with a sampler \(\sigma\) and for each ground operator \(\omega\) there is a ground sampler \(\sigma : X \rightarrow \Delta(\Theta)\), where \(\sigma(x) = \sigma(x, (o_1, \ldots, o_n))\).

With this abstract model, as well as the samplers, we can perform bilevel planning (See Appendix 3 for pseudocode and detailed description) to achieve particular goals. Bilevel planning begins by generating abstract plans of the form \(P = [\omega_0, \omega_1, \ldots, \omega_n]\). These plans specify sequences of predicted ground abstract states and ground operators that should achieve these states. Once we have an abstract plan that achieves the goal, we attempt to refine it into a series of ground skills \(\phi = (\phi_1, \ldots, \phi_k)\) using samplers associated with each ground operator in the plan. In particular, given the agent is currently in a state \(x_i\), \(\text{\texttt{abstract}}(x_i) = s_i\) and \(s_i \subseteq s_i\), and we use the known transition function \(f\) to simulate a ground skill \(\phi_{i + 1}\) execution that transitions us to a new state \(x_{i + 1}\), where \(s_{i + 1} = \text{\texttt{abstract}}(x_{i + 1})\). If the refinement is successful, then \(s_{i + 1} \subseteq s_{i + 1}\). If refinement fails more than a set number of times, we generate a new abstract plan and repeat the process. This approach to bilevel planning exactly follows the framework of search-then-sample Task and Motion Planning (TAMP) (Garrett et al. 2021).

### 3 Operators with Ignore Effects

In service of our objective of learning operators that enable efficient planning in complex domains where actions may cause a large number of symbolic effects, we extend traditional STRIPS operators to name effects that they will ignore. An operator with ignore effects is a tuple
Learning Operators with Ignore Effects

Given our training dataset \( D \), how can we learn operators with ignore effects that will lead to efficient planning on unseen evaluation tasks? A naive approach would explicitly search over sets of operators, evaluating each candidate set by planning in the training tasks (with some regularization to promote generalization). This approach is infeasible even for small problems; the search space is extremely large, and evaluating each candidate is slow because planning is computationally hard.

The core of our approach is to learn operators bottom-up from the training data, as in prior work (Silver et al. 2021b), but in the more general class of operators with ignore effects. To motivate our algorithm, consider a highly simplified problem setting where solving each task requires a single action. A demonstration for a task with goal \( g \) would be a single abstract transition \((s_0, \omega_0, s_1)\) where \( g \subseteq s_1 \). Though the set difference between \( s_0 \) and \( s_1 \) may be large, the only atoms that we need to model are \((s_1 \setminus s_0) \cap g\), that is, the goal atoms that were added during the transition. Any other atoms that were added or deleted could be safely ignored without compromising the ability to solve the task. Identifying these necessary and ignorable atoms is the basis for learning operators with ignore effects.

To extend this intuition to the two-step setting, where demonstrations are of the form \((s_0, \omega_0, s_1, \omega_1)\), where \( g \subseteq s_2 \), suppose that we have already learned an operator \( \omega_1 \) with preconditions \( P_1 \) that models the second step. We could similarly conclude that \((s_1 \setminus s_0) \cap P_1\) are necessary and other changes can be ignored, and learn operators for this first step, with the same procedure generalizing backward to the multistep case. This preimage backchaining is familiar from the regression planning literature (Pollock 1998; Weld 1994). However, in the context of operator learning, there is a paradox: we need to perform backchaining to learn operators, but we need operators to perform this backchaining.

In light of this, we propose a two-stage algorithm that learns a set of operators with ignore effects that enables bilevel planning to all goals in \( D \). At a high level, the first stage involves iterating preimage backchaining to induce a set of operators that capture only the necessary effects. The algorithm’s second stage then fills in each learned operator’s delete and ignore effects and adds new operators if necessary. The pseudocode for this algorithm is shown in Algorithm 1. We refer the reader to A.3 in the appendix for a detailed walkthrough on a particular example.

Algorithm 1 starts with an empty operator set. The entirety of the first phase is achieved by BACKCHAINONCE procedure (Algorithm 2). For each trajectory, this procedure maintains a set of necessary add effects that must be made true at this particular timestep in order to enable the rest of the learned operators to be formed into a plan that achieves the goal. The necessary add effects are initialized to be the goal atoms that change during the final transition. At each step proceeding backwards to the initial state, we either select (FINDBESTMATCHINGOP) a “best match” operator or, if no operator whose effects are a superset of these necessary add effects exists, invent (CREATENEWOP) a new one to satisfy the necessary add effects. We find best match operators using a heuristic that counts the number of effects in common between a candidate operator and the transition; see A.2 in the appendix for details. With an operator identified, we update the necessary add effects (Line 4, Algorithm 2) and continue. Importantly, every time a new operator is invented, we partition the entire set of transitions in \( D \) into partitions based on the current set of operators (i.e., the REPARTITION-DATA method) using the best match heuristic. Once our first phase has converged to a fixed operator set, we will have assigned an operator to each controller invocation in every training sequence.

The second phase of our algorithm involves passing over all transitions associated with each opera-
Algorithm 2: Pseudocode for the BACKCHAINONCE procedure used within Algorithm 1.

```plaintext
BACKCHAINONCE( D, Ω )
1 for TRAJ in D do
2   necessaryAtoms ← TRAJ.GOAL
3   for (s_i, o_i, s_i+1) in REVERSE(D) do
4     necessaryAddEffects ← (necessaryAtoms − s_i)
5     ω ← FINDBESTMATCHINGOP(Ω, necessaryAddEffects)
6     if ω is None then
7       ω ← CREATE-NEWOP(necessaryAddEffects)
8       Ω ← Ω ∪ {ω}
9     REPARTITIONDATA(Ω, D)
10    Ω ← RECOMPUTEPRECONDITIONS(Ω)
11   necessaryAtoms ← (necessaryAtoms − op.addEffects)
12   necessaryAtoms ← (necessaryAtoms ∪ op.preconditions)
13 return Ω
```

The algorithm for inducing that operator’s delete and ignore effects (COMPUTEDELANDIGNORE). This process might cause certain operators to interfere with the necessary effects of other operators (by ignoring these effects). Thus, we induce additional operators (FIXCHAINING) to fix these problems (for a detailed example of this, see A.3 in the appendix).

A critical aspect of this approach is our method for inducing the components of each operator. Most operators are created to satisfy certain necessary add effects (CREATENEWOP), so the operator’s add effects are set to be exactly these. Moreover, the operator’s controller ω.C is set to be the controller executed during the transition that corresponds to these necessary add effects. The only other situation in which we create new operators is to fix issues caused by interference between ignore effects and add effects (FIXCHAINING); here as well, we know exactly what the add effects and controller of the new operator should be. These add effects and controller specify the operator’s parameters ω.VAR. Following Silver et al. (2021b, 2022), we assume that the parameters are all the unique typed objects appearing in the union of the operator add effects and discrete controller parameter. The other components (i.e., preconditions, delete effects and ignore effects) of the operator are learned under this constraint (i.e., we assume that new parameters cannot be added to the operators when learning these components).

The preconditions, delete effects, and ignore effects are all induced via the transitions associated with an operator. Let D_w represent all the transitions (s_i, o_i, s_i+1) associated with a particular operator ω. Then, the operator’s precondition is set to be a lifted intersection over all initial states s_i in the dataset (RECOMPUTEPRECONDITIONS method): ω.P ← T(s_i,...) δ^{-1}(s_i)∀s_i ∈ D_w 1. Similarly, the delete effects are set to be a lifted union over all delete effects in the dataset: ω.E^- ← ∪T(s_i,...) δ^-1(s_i)∀(s_i, s_i+1) ∈ D_w. Now, the ignore effects are simply the set of all predicates with atoms that differ between a transition’s final state (s_i+1) and the state predicted by applying the operator to the transition’s initial state (s_i) (these are set in the COMPUTEDELANDIGNORE method).

### Learning Samplers

In addition to operators, we must also learn samplers to refine operators during planning. We directly adapt existing approaches (Chitnis et al. 2022; Silver et al. 2022) to accomplish this and learn samplers of the following form: σ(x, o_1,..., o_k) = s_σ(x[o_1] ⊕ ... ⊕ x[o_k]), where x[o] denotes the feature vector for o in x, the ⊕ denotes concatenation, and s_σ is the model to be learned. Specifically, we treat the problem as one of supervised learning from the transition data associated with each of the learned operators: D_w. Recall that for every transition (s_i, o_i, s_i+1) associated with a particular operator, there is a corresponding low-level transition (τ_i, δ_i) belonging to the transition τ_i. To create a datapoint that can be used for supervised learning for the associated sampler, we reuse the substitution δ_i found during precondition learning to create an input vector x[δ_i(v_1)] ⊕ ... ⊕ x[δ_i(v_k)], where (v_1,..., v_k) = PAR. The corresponding output for supervised learning is the continuous parameter vector θ in the ground skill φ. For more details see A.5 in the appendix.

### Algorithm Properties

We note that the overall LearnOpsByBackchaining procedure is guaranteed to terminate. At a high level, this is because the operator set Ω only ever grows in size throughout the procedure, and new operators added to this set must be unique. Since there are only a finite number of predicates and objects in the training set, there is an upper bound to the number of operator hypotheses that might be considered. In the worst case, the algorithm’s operator set Ω will hit this bound and then terminate.

Additionally, we note that our algorithm produces operators that are sound under ignore effects over the data in the following sense: for any training task, there exists at least one plan using the learned operators Ω such that the sequence of controllers associated with the operators in this plan is exactly the sequence of controllers executed in the demonstration. This stems from the fact that, when the algorithm terminates, FINDBESTMATCHINGOP method must have returned an operator from Ω for every transition in the training data 1.

### 5 Experiments

Our experiments were designed to answer the following questions: (Q1) To what extent is our algorithm able to learn 2

---

1Recall that δ is a map between an operator’s parameters to particular objects. For the definition, see Section 2.2.
2For full proofs and explanations see A.4 in the appendix.
operators that lead to efficient planning, and how does planning efficiency compare to existing methods? (Q2) How does the efficiency of operators learned by our algorithm change with the size of the training dataset it is given?

5.1 Experimental Setup

We evaluate six methods across six robotic planning environments of varying difficulty. All reported experiments were conducted on a quad-core Intel Xeon Platinum 8260 processor, and all results are averaged over 10 random seeds, which vary the training and evaluation tasks, random initializations during learning, and tiebreaking during planning. For each seed and task, we sample a set of 50 evaluation tasks from the task distribution $T$. The evaluation tasks all have a larger number of objects, different initial states, and harder variations of goals than were seen during training. Our key measure of efficient planning is success rate given a 10-second timeout. For additional results about the efficiency of learning itself, see A.8 in the appendix.

Environments

Our environments were selected to represent a range of realistic, complex tasks that evince different situations in which learning operators with ignore effects is desirable, as well as a set of environments that can be well-solved using STRIPS operators without ignore effects. We provide the high-level details behind our experimental environments here (see A.6 in the appendix for further details).

- **Screws**: an implementation of the environment depicted in Figure 1. The agent controls a crane and is tasked with picking up a particular screw and placing it in a receptacle for later use. The environment is challenging because problems generally contain a large number of screws strewn together in piles such that every attempt to pick up a particular screw will result in a large number of different screws being picked up.

- **Repeated NextTo**: a simple environment where the robot is tasked with collecting a number of objects cluttered along a 1D line. To pick up an object, a robot must be 'next to' it. However, the large number of objects makes it such that the robot is always next to a different number of objects after it moves.

- **Satellites Simple**: an environment inspired by the benchmark Satellites domain from Bacchus (2001). The agent controls a fleet of satellites that must calibrate themselves against specific targets, then coordinate to take readings of a particular object. Each satellite’s sensors have a fan-shaped area and distance range in which they can take readings or be calibrated. This environment does not require operators with ignore effects.

- **Satellites**: an environment that is the same as 'Satellites Simple’, except that there are multiple objects that readings must be taken from. Thus, when a satellite turns or moves to a particular object, there might be a large number of different objects that are within its sensor area.

- **Painting**: a challenging robotics environment used by Silver et al. (2021b, 2022). A robot in 3D must pick, wash, dry, paint, and then place various objects into either a box or a shelf. This environment does not require operators with ignore effects.

- **RNT Painting**: a challenging robotics environment that combines the Repeated NextTo and Painting environments; the task is the same as in Painting, except that the robot can be next to multiple objects at a time. Thus, every time it moves, the robot will find itself next to a different number of objects.

Baselines

We provide an overview of the various baselines used in our experiments (for details, see A.7 in the appendix). The first three approaches attempt to explicitly learn operators by optimizing prediction error in some fashion, while the latter two attempt to directly learn policies...
from the demonstrations.

- **Cluster and Intersect**: This is the operator learning approach used by Silver et al. (2022). It learns STRIPS operators by attempting to induce a different operator for every set of unique lifted effects.
- **LOFT**: This is the operator learning approach used by Silver et al. (2021b); it also learns STRIPS operators by optimizing prediction error.
- **Cluster and Intersect with Ignore Effects (CI + IE)**: This approach is a variant of Cluster and Intersect inspired by LOFT that is capable of learning operators with ignore effects. In particular, it first runs Cluster and Intersect, then attempts to induce ignore effects by performing a hill-climbing search over possible choices of ignore effects using prediction error as the metric to be optimized.
- **GNN Shooting**: This approach trains a graph neural network (GNN) (Battaglia et al. 2018) policy. The GNN takes in the current state \( x \), abstract state \( s \), and goal \( g \). It outputs an action via a one-hot vector over \( C \), one-hot vectors over all objects at each discrete argument position, and a vector of continuous arguments. We train the GNN using behavior cloning on the dataset \( D \). At evaluation time, we sample trajectories by treating the GNN’s output continuous arguments as the mean of a Gaussian with fixed variance. We use the known transition function \( f \) to check if the goal has been achieved, and resample accordingly.
- **GNN Model-Free**: A baseline that uses the same trained GNN as above, but at evaluation time, directly executes the policy instead of checking execution using \( f \).

All approaches are trained with the same demonstration data and use the same predicates. All approaches also use the transition function \( f \), except for GNN Model-Free, which is completely model-free.

### 5.2 Results and Analysis

Figure 2 shows success rate as a function of the size of the training set for all our environments. Our method is able to achieve a near 100% success rate on the testing tasks in all the environments after training on 50 demos. Moreover, at any training dataset size, it outperforms or matches the performance of all baseline methods within the 10 second planning timeout (Q1)\(^3\). It is able to do this not only for environments where operators with ignore effects can lead to more efficient and effective planning (top row), but also in environments where such operators are unnecessary and approaches that learn standard STRIPS operators perform just as well (bottom row). Additionally, our approach’s success rate improves with more data (Q2), especially in the more challenging Satellites and RNT Painting environments.

### 6 Related Work

Our work continues a long line of research in learning operators for planning (Drescher 1991; Schmill, Oates, and Cohen 2000; Amir and Chang 2008; Krüger et al. 2011; Lang, Toussaint, and Kersting 2012; Mourao et al. 2012; Pasula, Zettlemoyer, and Kaelbling 2007; Rodrigues et al. 2011; Cresswell, McCluskey, and West 2013; Aineto, Jiménez, and Onaindia 2018); see Arora et al. (2018) for a recent survey. Most of this prior work focuses on finding an accurate model in a domain with discrete state and action spaces.

Other work has considered learning symbolic planning models in continuous environments (Jetchev, Lang, and Toussaint 2013; Ugur and Pater 2015; Ahmetoglu et al. 2020; Asai and Fukunaga 2018; Bonet and Geffner 2020; Asai and Muise 2020). Several of these works explicitly Umili et al. (2021); Konidaris, Kaelbling, and Lozano-Pérez (2018); Curtis et al. (2022) use a bisimulation or prediction error objective to learn predicates, which we seek to improve upon for the operator learning problem.

Our efforts are most directly inspired by LOFT and learning Nero-Symbolic Relational Transition Models (Silver et al. 2021b; Chitnis et al. 2022), which are approaches that learn operators and samplers (among other components) that can be used for bilevel planning in continuous environments by explicitly optimizing prediction error and attempting to model each transition in the data respectively. In fact, this work was originally motivated by our failed attempt to apply these prior methods to a mobile manipulation environment with a NextTo predicate (Srivastava et al. 2021). We include LOFT and ‘Cluster and Intersect’ as baselines representative of these approaches in our experiments. Additionally, our proposal for extending STRIPS operators to handle non-deterministic effects is not new: we take inspiration from Marthi, Russell, and Wolfe (2007)’s NCSTRIPS. Our approach is notably different from theirs in that we can declare all atoms associated with a particular predicate to be non-deterministic.

The bilevel planner used in our work can be viewed as a search-then-sample solver for task and motion planning (TAMP) (Srivastava et al. 2014; Dantam et al. 2016; Garrett et al. 2021). To that end, our work also contributes to a recent line of work on learning for TAMP. Other efforts in this line include sampler learning (Chitnis et al. 2016; Mandleika et al. 2019; Wang et al. 2021; Ortiz-Haro et al. 2022), heuristic learning (Shen, Trevizan, and Thiébaux 2020; Kim and Shimaniu 2019; Paxton et al. 2017), and abstract plan feasibility estimation (Driess, Ha, and Toussaint 2020; Noseworthy et al. 2021).

### 7 Conclusion and Future Work

In this work, we introduced operators with ignore effects as a generalization of STRIPS operators that can help enable more efficient bilevel planning in robotics domains. We also developed an algorithm that is able to efficiently learn these operators from demonstrations. Experiments on a number of challenging robotics domains illustrate that our approach is capable of learning operators that lead to efficient and effective planning on problems unseen during training.

Important next steps include integrating this method with strategies for searching the space of predicates, for learning policies for the controllers, and for handling noise in the training trajectories. We believe that pursuing these

---

\(^3\)For a detailed comparison of the nodes expanded during high-level planning, see A.8 in the appendix.
steps will yield important progress toward solving sparse-feedback, long-horizon decision-making problems at scale.

Acknowledgements
We gratefully acknowledge support from NSF grant 1723381; from AFOSR grant FA9550-22-1-0249; from ONR grant N00014-18-1-2847; from MIT-IBM Watson Lab; and from the MIT Quest for Intelligence. Nishanth, Willie, and Tom are supported by NSF Graduate Research Fellowships. Any opinions, findings, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors. We thank Aidan Curtis for helpful comments on an earlier draft of this work.

References
Ahmetoglu, A.; Seker, M. Y.; Piater, J.; Oztop, E.; and Ugur, E. 2020. Deepsym: Deep symbol generation and rule learning from unsupervised continuous robot interaction for planning. arXiv preprint arXiv:2012.02532.

Ainetó, D.; Jiménez, S.; and Onaindia, E. 2018. Learning STRIPS action models with classical planning. In The International Conference on Automated Planning and Scheduling (ICAPS).

Amir, E.; and Chang, A. 2008. Learning partially observable deterministic action models. The Journal of Artificial Intelligence Research (JAIR).

Arora, A.; Fiorino, H.; Pellier, D.; Métivier, M.; and Pesty, S. 2018. A review of learning planning action models. The Knowledge Engineering Review.

Asai, M.; and Fukunaga, A. S. 2018. Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary. In The AAAI Conference on Artificial Intelligence (AAAI).

Asai, M.; and Muise, C. 2020. Learning neural-symbolic descriptive planning models via cube-space priors: The voyage home (to STRIPS). arXiv preprint arXiv:2004.12850.

Bacchus, F. 2001. AIPS 2000 planning competition: The Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems. AI Magazine.

Battaglia, P. W.; Hamrick, J. B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; et al. 2018. Relational inductive biases, deep learning, and graph networks. arXiv preprint arXiv:1806.01261.

Bonet, B.; and Geffner, H. 2020. Learning first-order symbolic representations for planning from the structure of the state space. In The European Conference on Artificial Intelligence (ECAI).

Chitnis, R.; Hadfield-Menell, D.; Gupta, A.; Srivastava, S.; Groshev, E.; Lin, C.; and Abbeel, P. 2016. Guided search for task and motion plans using learned heuristics. In The IEEE International Conference on Robotics and Automation (ICRA).

Chitnis, R.; Silver, T.; Tenenbaum, J. B.; Lozano-Pérez, T.; and Kaelbling, L. P. 2022. Learning Neuro-Symbolic Relational Transition Models for Bilevel Planning. In The IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).

Cresswell, S. N.; Mccluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using LOCM. The Knowledge Engineering Review.

Curtis, A.; Silver, T.; Tenenbaum, J. B.; Lozano-Pérez, T.; and Kaelbling, L. 2022. Discovering state and action abstractions for generalized task and motion planning. In The AAAI Conference on Artificial Intelligence (AAAI).

Dantam, N. T.; Kingston, Z. K.; Chaudhuri, S.; and Kavraki, L. E. 2016. Incremental task and motion planning: A constraint-based approach. In Robotics: Science and Systems (R:SS).

Dean, T.; and Givan, R. 1997. Model Minimization in Markov Decision Processes. In Conference on Innovative Applications of Artificial Intelligence (IAAI).

Drescher, G. L. 1991. Made-up minds: a constructivist approach to artificial intelligence. MIT press.

Driess, D.; Ha, J.-S.; and Toussaint, M. 2020. Deep Visual Reasoning: Learning to Predict Action Sequences for Task and Motion Planning from an Initial Scene Image. In Robotics: Science and Systems (R:SS).

Garrett, C. R.; Chitnis, R.; Holladay, R.; Kim, B.; Silver, T.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Integrated task and motion planning. Annual review of control, robotics, and autonomous systems.

Jetchev, N.; Lang, T.; and Toussaint, M. 2013. Learning grounded relational symbols from continuous data for abstract reasoning. In ICRA Workshop on Autonomous Learning.

Kim, B.; and Shimanuki, L. 2019. Learning value functions with relational state representations for guiding task-and-motion planning. Conference on Robot Learning (CoRL).

Konidaris, G.; Kaelbling, L. P.; and Lozano-Pérez, T. 2018. From skills to symbols: Learning symbolic representations for abstract high-level planning. The Journal of Artificial Intelligence Research (JAIR).

Krüger, N.; Geib, C.; Piater, J.; Petrick, R.; Steedman, M.; Wörgötter, F.; Ude, A.; Asfour, T.; Kraft, D.; Omrèn, D.; et al. 2011. Object–action complexes: Grounded abstractions of sensory–motor processes. Robotics and Autonomous Systems.

Lang, T.; Toussaint, M.; and Kersting, K. 2012. Exploration in relational domains for model-based reinforcement learning. The Journal of Machine Learning Research (JMLR).

Mandalika, A.; Choudhury, S.; Salzman, O.; and Srinivasa, S. 2019. Generalized lazy search for robot motion planning: Interleaving search and edge evaluation via event-based toggles. In The International Conference on Automated Planning and Scheduling (ICAPS).

Marthi, B.; Russell, S. J.; and Wolfe, J. A. 2007. Angelic Semantics for High-Level Actions. In The International Conference on Automated Planning and Scheduling (ICAPS), 232–239.
Mourao, K.; Zettlemoyer, L. S.; Petrick, R.; and Steedman, M. 2012. Learning STRIPS operators from noisy and incomplete observations. arXiv preprint arXiv:1210.4889.

Noseworthy, M.; Moses, C.; Brand, I.; Castro, S.; Kaelbling, L.; Lozano-Pérez, T.; and Roy, N. 2021. Active Learning of Abstract Plan Feasibility. In Robotics: Science and Systems (R:SS).

Ortiz-Haro, J.; Ha, J.-S.; Driess, D.; and Toussaint, M. 2022. Structured deep generative models for sampling on constraint manifolds in sequential manipulation. In Conference on Robot Learning (CoRL).

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2007. Learning symbolic models of stochastic domains. The Journal of Artificial Intelligence Research (JAIR).

Paxton, C.; Raman, V.; Hager, G. D.; and Kobilarov, M. 2017. Combining neural networks and tree search for task and motion planning in challenging environments. In The IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).

Pollock, J. L. 1998. The logical foundations of goal-regression planning in autonomous agents. Artificial Intelligence.

Rodrigues, C.; Gérard, P.; Rouveirol, C.; and Soldano, H. 2011. Active learning of relational action models. In International Conference on Inductive Logic Programming.

Schmill, M. D.; Oates, T.; and Cohen, P. R. 2000. Learning Planning Operators in Real-World, Partially Observable Environments. In The International Conference on Artificial Intelligence Planning Systems (AIPS).

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning domain-independent planning heuristics with hypergraph networks. In The International Conference on Automated Planning and Scheduling (ICAPS).

Silver, T.; Chitnis, R.; Kumar, N.; McClinton, W.; Lozano-Pérez, T.; Kaelbling, L. P.; and Tenenbaum, J. 2022. Invent- ing relational state and action abstractions for effective and efficient bilevel planning. In The Multi-disciplinary Conference on Reinforcement Learning and Decision Making (RLDM).

Silver, T.; Chitnis, R.; Tenenbaum, J.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021a. Learning symbolic operators for task and motion planning. In 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).

Silver, T.; Chitnis, R.; Tenenbaum, J.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021b. Learning Symbolic Operators for Task and Motion Planning. In The IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).

Srivastava, S.; Fang, E.; Riano, L.; Chitnis, R.; Russell, S.; and Abbeel, P. 2014. Combined task and motion planning through an extensible planner-independent interface layer. In The IEEE International Conference on Robotics and Automation (ICRA).

Srivastava, S.; Li, C.; Lingelbach, M.; Martín-Martín, R.; Xia, F.; Vainio, K. E.; Lian, Z.; Gokmen, C.; Buch, S.; Liu, K.; Savarese, S.; Gweon, H.; Wu, J.; and Fei-Fei, L. 2021. BEHAVIOR: Benchmark for Everyday Household Activities in Virtual, Interactive, and Ecological Environments. In Conference on Robot Learning (CoRL).

Ugur, E.; and Piater, J. 2015. Bottom-up learning of object categories, action effects and logical rules: From continuous manipulative exploration to symbolic planning. In The IEEE International Conference on Robotics and Automation (ICRA).

Umili, E.; Antonioni, E.; Riccio, F.; Capobianco, R.; Nardi, D.; and De Giacomo, G. 2021. Learning a Symbolic Planning Domain through the Interaction with Continuous Environments. ICAPS PRL Workshop.

Wang, Z.; Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Learning compositional models of robot skills for task and motion planning. The International Journal of Robotics Research (IJRR).

Weld, D. S. 1994. An Introduction to Least Commitment Planning. AI Magazine.
A Appendix

A.1 Detailed Planning Algorithm Descriptions

**Algorithm 3:** This is the pseudocode for our search-then-sample bilevel planning algorithm. The inputs are an objects $\mathcal{O}$, initial state $x_0$, goal $g$, predicates $\Psi$, operators $\Omega$, and samplers $\Sigma$; the output is a plan $\pi$. Our outer loop GENABSTRACTPLAN generates plans in the abstract state and action spaces, which guide our inner loop. This inner loops runs REFINE which samples continuous parameters for each plan from our samplers $\Sigma$ to concretize each abstract plan $\hat{\pi}$ into a plan $\pi$. If REFINE succeeds, then the found plan $\pi$ is returned as the solution; if REFINE fails, then GENABSTRACTPLAN continues.

**Algorithm 4:** This is GENABSTRACTPLAN which finds a task plan by creating operators for all possible groundings then uses $A^*$ search to find the top $n_{\text{abstract}}$ plans. It returns a list of plans $\hat{\pi}$.

**Algorithm 5:** This is REFINE which turns a task plan $\hat{\pi}$ into a sequence of ground skills. It gets the state and operators from $\hat{\pi}$ and adds the controller with newly sampled continuous parameters to $\pi$. After this it checks to see if the added controller is initiable from the current state in the plan and we simulate the skill execution to verify if it reached the expected state we predicted next in $\hat{\pi}$. If the controller is not initiable or fails the expected atoms check we backtrack and resample a new continuous parameter for this controller until either we reach the max number of samples or we successfully refine our final controller.

```plaintext
A.2 Data Partitioning Heuristic

A key component of our algorithm is the process by which we associate transitions with operators. This is not only used to partition the dataset amongst operators in the REPARTITIONDATA method, but also to compute the best matching operator for a particular datapoint with certain necessary add effects in the FINDBESTMATCHINGOP method. Given a transition $(s_i, \phi_i, s_{i+1})$ and operator set $\Omega$, we perform this association in two steps. First, we find all $\omega \in \Omega$ that might reasonably match the transition. We do this by checking that $\omega.P$ are true in $s_i$, $\omega.C$ matches the controller associated with $\phi_i$, the set of atoms $F(s_i, \omega)$ is true in $s_{i+1}$, and finally, if we’ve previously computed the necessary add effects for this transition, that these necessary add effects are a subset of $\omega.E^\dagger$. For each $\omega \in \Omega$ that satisfies these conditions, we compute a score using the formula shown in equation
```
Even, which attempts to measure how well a particular operator ‘fits’ the transition. Intuitively, the formula measures the difference between the operator’s effects and the observed effects; the closer these match, the lower the score given.

The one caveat to this approach is that our algorithm (specifically the FIXCHAINING procedure) induces operators that have the same lifted atom in both their precondition and add effects (see Section A.3 for a concrete example of such operators). These particular sections do not change when the operator is applied, and so we remove such ‘keep’ effects from consideration in our heuristic. Moreover, since our algorithm only induces such operators when it detects they are necessary, we score operators with keep effects lower than those without.

\[
\begin{align*}
\text{keep \_ effects} &= \text{op \_ preconditions} \cap \text{op \_ add \_ effects} \\
\text{nonkeep \_ effects} &= \text{op \_ add \_ effects} - \text{keep \_ effects} \\
\text{score} &= ||(\text{nonkeep \_ effects} - \text{transition \_ add \_ effects})|| + \\
&\quad ||(\text{transition \_ add \_ effects} - \text{nonkeep \_ effects})|| + \\
&\quad ||(\text{op \_ delete \_ effects} - \text{transition \_ delete \_ effects})|| + \\
&\quad ||(\text{transition \_ delete \_ effects} - \text{op \_ delete \_ effects})|| - \\
&\quad \text{keep \_ effects} \\
\end{align*}
\]

Once all eligible operators have been scored, we simply pick the lowest-scoring operator to associate with this transition. If multiple operators achieve the same score, we break ties arbitrarily.

#### A.3 Extended Example of Operator Learning Algorithm

Here, we provide an extended example of our learning algorithm on a small dataset to supplement Section 4.

The example demonstration dataset is shown in Figure 3. These demonstrations occur within a simple domain we call Machines. There are a specific set of machines (m1, m2, and m3) in a factory, each of which can be turned on or off, configured or de-configured, and then run. Any machine can be turned on and configured at any time (i.e., there are no preconditions for either of these actions), however, configuring a particular machine may cause some other machines to turn on or off (note that which specific machines get turned on/off is deterministic in every task). The agent’s goal is to run a particular machine from various different initial states.

Following, Algorithm 1, the operator set \( \Omega \) will be initialized to be empty, and the BACKCHAINONCE procedure will then be called on the operator set and the demonstrations.

**BACKCHAINONCE Procedure** This procedure will first consider the first demonstration in reverse order. Initially, the ‘necessary atoms’ will be set to the trajectory’s goal: Has Been Run (m1). Since this atom is absent in the penultimate state, the ‘necessary add effects’ will also be Has Been Run (m1). Since the current operator set is empty, FINDBESTMATCHINGOP will return *None* for these ‘necessary add effects’. The procedure will then create a new operator associated with the Run skill that has a lifted add effect Has Been Run (?m1), where

\(?m1 \) refers to the variable of type *machine* that will be in the operator’s parameters. Given this operator, the REPARTITIONDATA procedure will associate both transitions that use the Run skill (i.e., transitions 1.3 and 2.3) with this operator. The REcomputePreconditions procedure will then compute the preconditions of this sole operator by taking the lifted intersection of the abstract initial states of these associated transitions. In this case, this will be \{ Configured (?m1), On (?m1) \} \cup \{ On (?m1), Configured (?m1) \}, which is simply \{ Configured (?m1), On (?m1) \}. This operator is shown below:

**Run**

Parameters: (?m1)
Preconditions: [Configured (?m1), On (?m1)]
Add Effects: [HasBeenRun (?m1)]
Delete Effects: []
Ignore Effects: []
Skill: Run (?m1)
Associated Transitions: [1.3, 2.3]

The ‘necessary atoms’ will then be updated to remove Has Been Run (m1) and add in the atoms from the operator’s preconditions, specifically: Configured (m1), On (m1).

By following this sequence of steps for each of the transitions in the given demonstrations, we can see that our final operator set at the end of the procedure will consist of: \(^4\)

**Run**

Parameters: (?m1)
Preconditions: [Configured (?m1), On (?m1)]
Add Effects: [HasBeenRun (?m1)]
Delete Effects: []
Ignore Effects: []
Skill: Run (?m1)
Associated Transitions: [1.3, 2.3]

**Configure**

Parameters: (?m1)
Preconditions: []
Add Effects: [Configured (?m1)]
Delete Effects: []
Ignore Effects: []
Skill: Configure (?m1)
Associated Transitions: [1.1, 2.2]

**TurnOn**

Parameters: (?m1)
Preconditions: []
Add Effects: [On (?m1)]
Delete Effects: []
Ignore Effects: []
Skill: TurnOn (?m1)
Associated Transitions: [1.2, 2.1]

\(^4\) In fact, these operators will be induced after passing through just the first demonstration. They will not change as the algorithm passes backward through the second demonstration.
To do this, it must be necessary. We will refer to specific transitions within these demonstrations by the following indexing scheme: demonstration index:i;transition index:i. For example, 1.1 will refer to the first transition in the first demonstration.

**COMPUTEDELANDIGNORE Procedure**  This procedure simply passes through each of the transitions associated with the current operators to compute delete and ignore effects. As an example, let’s consider the transitions associated with the Configure operator (i.e 1.1 and 2.2):

\[
\begin{align*}
\{\} & \xrightarrow{\text{Configure}(m_1)} \{\text{Configured}(m_1), \text{On}(m_2)\} \\
\{\text{On}(m_3)\} & \xrightarrow{\text{Configure}(m_1)} \{\text{On}(m_3), \text{On}(m_1)\}
\end{align*}
\]

\[
\begin{align*}
\{\} & \xrightarrow{\text{Configure}(m_1)} \{\text{Configured}(m_1), \text{On}(m_2)\} \\
\{\text{On}(m_3)\} & \xrightarrow{\text{Configure}(m_1)} \{\text{On}(m_3), \text{On}(m_1)\}
\end{align*}
\]

Here, only the second transition (i.e 2.2) has a delete effect (specifically On(m3)). However, the procedure will not induce a corresponding delete effect for this atom because that would require adding another variable (?m2 corresponding to m3 in this transition) to our operator’s parameters. Thus, no delete effects are induced. Given this, the On predicate must be added to the operator’s ignore effects, since the existing effects do not correctly predict how atoms related to this predicate change in all of the transitions.

By following these steps for each of the 3 operators, we are left with the following set at the end of the procedure:

**Run**
Parameters: (?m1)
Preconditions: [Configured(?m1), On(?m1)]
Add Effects: [HasBeenRun(?m1)]
Delete Effects: []
Ignore Effects: []
Skill: Run(?m1)
Associated Transitions: [1.1, 2.2]

**Configure**
Parameters: (?m1)
Preconditions: []
Add Effects: [Configured(?m1)]
Delete Effects: []
Ignore Effects: [On]
Skill: Configure(?m1)

**TurnOn**
Parameters: (?m1)
Preconditions: []
Add Effects: [On(?m1)]
Delete Effects: []
Ignore Effects: []
Skill: TurnOn(?m1)
Associated Transitions: [1.2, 2.1]

**FIXCHAINING Procedure**  This procedure checks to see whether the ignore effects induced by the COMPUTEDELANDIGNORE procedure might prevent the agent from replicating the demonstrations using its operators. In particular, it checks whether ignore effects interfere with the necessary atoms for any of the transitions. In our example, this occurs with the Configure operator and transition 2.2. In particular, since our learned Configure operator has On in its ignore effects, then if it is executed in the same situation as transition 2.2 (i.e, when On(m1) is true), then the agent would be unable to use its Run operator afterwards to achieve the goal (since On(m1) would be ignored and dropped from the state after executing the Configure(m1) operator).

To fix this issue, the procedure will create a new operator that includes an additional precondition and add effect. In the case of our example, only one such operator is necessary. The final set of operators returned by this procedure is shown below:

**Run**
Parameters: (?m1)
Preconditions: [Configured(?m1), On(?m1)]
Add Effects: [HasBeenRun(?m1)]
Delete Effects: []
Ignore Effects: []
Skill: Run(?m1)
Associated Transitions: [1.3, 2.3]

**Configure**
Parameters: (?m1)
Preconditions: []
Add Effects: [Configured(?m1)]
Delete Effects: []
Ignored Effects: [On]
Skill: Configure(?m1)
Associated Transitions: [1.1, 2.2]

**Configure-with-On**

Parameters: (?m1)
Preconditions: [On(?m1)]
Add Effects: [Configured(?m1), On(?m1)]
Delete Effects: []
Ignored Effects: [On]
Skill: Configure(?m1)
Associated Transitions: [2.2]

**TurnOn**

Parameters: (?m1)
Preconditions: []
Add Effects: [On(?m1)]
Delete Effects: []
Ignored Effects: []
Skill: TurnOn(?m1)
Associated Transitions: [1.2, 2.1]

This set will not change on the subsequent iteration, and thus, this is the final set that will be returned.

### A.4 Algorithm Properties

#### Algorithm Termination

Let us first show that the **BACKCHAIN** procedure must terminate. Notice that the operator set $\Omega$ only ever grows in size (lines 7 and 8). Moreover, new operators added to this set must be unique (i.e., they must have different add effects or preconditions from any existing operator) since we only create an operator if $\Omega$ does not already contain an operator that satisfies certain necessary add effects, preconditions or controller (i.e., **FindBestMatchingOperator**($\Omega$, necessary_add_effects) yields None). The number of possible necessary add effects is finite because necessary add effects can only ever be a subset of a transition’s add effects, and there are a finite number of transition add effects in our training data. The number of preconditions is also finite because preconditions are induced by taking an intersection over some set of transitions with finite atoms, and the number of transitions is finite. Finally, the number of controllers is known to be finite. Thus, the **FindBestMatchingOperator** procedure will in the worst be called on the largest possible $\Omega$, such that it will never return ‘None’ for any necessary add_effects. At this point, the procedure must terminate.

We can now use a similar argument to show that the overall **LEARNOPSBYBACKCHAIN** procedure will terminate. Since there are a finite number of preconditions and add effects, there must also be a finite number of possible keep effects. Moreover, the **FixChaining** procedure will only add a new operator with keep effects to $\Omega$ if such an operator with keep effects does not already exist. Thus, the combination of the **BACKCHAIN** and **FixChaining** procedures will eventually add all possible operators with keep effects to $\Omega$. At this point, the **BACKCHAIN** procedure must terminate without making any modifications to $\Omega$. When this occurs, the **ComputeDelAndIgnore** method also cannot make any modifications to $\Omega$ (since the partitioning of the data amongst operators would not have
Soundness over Ignore Effects This is the property that, for any training task, there exists at least one plan using the learned operators \( \Omega \) such that the sequence of controllers associated with the operators in this plan is exactly the sequence of controllers executed in the demonstration.

To see that this holds, note that for the BACKCHAIN-ONCE procedure to terminate, the FINDBESTMATCHINGOP method must have returned an operator from the learned operator set for every single transition within every trajectory of our training set. This requires, among other things, that the operator’s controller exactly match the controller used in the transition \( \omega.C = \phi.\text{controller} \), that the operator’s precondition holds in the transition’s pre-image state \( s_i.\omega.P \subset s_i \), and that the operator’s add effects hold in the image state \( s_{i+1}.\omega.E^+ \subset s_{i+1} \). This guarantees that a plan exists that exactly matches any training trajectory. Moreover, for the overall LEARNOPSBYBACKCHAINING procedure to terminate, none of the methods called after BACKCHAIN-ONCE can have modified \( \Omega \) in any way. Finally, deleting any operators that have no data associated with them (i.e., calling the PRUNENULLDATAOPERATORS method) cannot affect soundness because only operators with data could possibly have been selected by the FINDBESTMATCHINGOP method.

Anytime Pruning of Null Data Operators In the procedure as illustrated in Algorithm 1, we only prune out operators that do not have any data associated with them after the main while loop has terminated. However, we note here that we can remove such operators from the current operator set (\( \Omega_{\text{current}} \)) at any time during the algorithm.

This property arises because the amount of data associated with an operator will only decrease over time. To see this, note that (1) the number of operators in the current set only increases over time, and (2) data is assigned to the ‘best matching’ operator as judged by our heuristic in Equation 1. Given a particular operator \( \omega \) at some iteration \( i \) of the algorithm, suppose there are \( d \) datapoints from the training demonstration trajectories associated with it. During future (i.e., \( i > k \) ) iterations of the algorithm, new operators will have been added to the current operator set. For any of the \( d \) datapoints associated with \( \omega \), these new operators can either be a worse match (in which case, the datapoint will remain associated with \( \omega \)), or a better match (in which case, the datapoint will become associated with the new operator). Thus, for any operator \( \omega \), once there is no longer any data associated with it, there will never be any data associated with it, and it will simply be pruned after the while loop terminated.

As a result, we can prune operators from our current set whenever there is no data associated with them. We do this in our implementation, since it improves the algorithm’s runtime.

A.5 Learning Samplers
Following previous work by Silver et al. (2022) and Chitnis et al. (2022), we learn two neural networks to parameterize each sampler. The first neural network takes in \( x_{[0]} \oplus \cdots \oplus x_{[k]} \) and regresses to the mean and covariance matrix of a Gaussian distribution over \( \theta \). We assume that the desired distribution has nonzero measure, but the covariances can be arbitrarily small in practice. To improve the representational capacity of this network, we learn a second neural network that takes in \( x_{[0]} \oplus \cdots \oplus x_{[k]} \) and \( \theta \), and returns true or false. This classifier is then used to rejection sample from the first network. To create negative examples, we use all transitions \( \tau' \) such that the controller in \( \tau' \) matches that in \( \text{CON} \), but the effects in \( \tau' \) are different from \( (E^+, E^-) \).

A.6 Additional Experiment Details
Here we provide detailed descriptions of each of experiment environments. See Section 5.1 for high-level descriptions and the accompanying code for implementations.

Screws Environment Details
- **Types:**
  - The screw type has features \( x, y, \text{held} \).
  - The receptacle type has features \( x, y \).
  - The gripper type has features \( x, y \).
- **Action space:** \( \mathbb{R}^2 + \{0,1\} \). An action \((dx, dy, \text{magnetization})\) is a delta on the gripper and whether it is unmagnetized or magnetized, respectively.
- **Predicates:**
  - Pickable(\( ?x_0:\text{gripper}, ?x_1:\text{receptacle} \)), AboveReceptacle(\( ?x_0:\text{gripper}, ?x_1:\text{receptacle} \)), HoldingScrew(\( ?x_0:\text{gripper}, \text{HoldingScrew}(?x_0:gripper, ?x_1:screw), \text{ScrewInReceptacle}(?x_0:screw, ?x_1:receptacle) \)).
- **Controllers:**
  - MoveToScrew(\( ?x_0: \text{gripper}, \text{ScrewInReceptacle}(?x_0:screw, ?x_1:receptacle) \)): moves the gripper to be Near the screw \( ?x_1 \).
  - MoveToReceptacle(\( ?x_0: \text{gripper}, \text{AboveReceptacle}(?x_0:gripper, ?x_1:receptacle) \)): moves the gripper to be AboveReceptacle(\( ?x_0:gripper, ?x_1:receptacle) \).
  - MagnetizeGripper(\( ?x_0: \text{gripper} \)): Magnetizes the gripper at the current location, which causes all screws that the gripper is Near to be held by the gripper.
  - DemagnetizeGripper(\( ?x_0: \text{gripper} \)): Demagnetizes the gripper at the current location, which causes all screws that are being held by the gripper to fall.
- **Goal:** The agent must make \( \text{ScrewInReceptacle}(?x_0:screw, ?x_1:receptacle) \) true for a particular screw that varies per task.

Repeated NextTo Environment Details
- **Types:**
  - The robot type has features \( x \).
Figure 4: Environments. Visualizations for our Screws, Satellites, and RNT Painting environments.

- The dot type has features $x$, grasped.
- **Action space**: $\mathbb{R}^3$. An action (move_or_grasp, move_loc, grasp_loc) represents whether to move (less than 0.5) or grasp (greater than 0.5), the new robot location (if first dim is move), and the normalized location of dot to grasp (if second dim is grasp) where the normalization is between the upper and lower bound of the environment $[\text{env}_{\text{lb}}, \text{env}_{\text{ub}}] \rightarrow [0, 1]$.
  - **Predicates**: \text{NextTo}(\text{x}0: \text{robot}, \text{x}1: \text{dot}), \text{NextToNothing}(\text{x}0: \text{robot}), \text{Grasped}(\text{x}0: \text{robot}, \text{x}1: \text{dot}).
  - **Controllers**:
    - \text{MoveGrasp}(\text{x}0: \text{robot}, \text{x}1: \text{dot}, [\text{move_or_grasp}, x])]: A single controller that performs both moving and grasping. If move_or_grasp < 0.5, then the controller moves the robot to a continuous position $y$. Else, the controller grasps the dot $\text{x}1$ if it is within range.
  - **Goal**: The agent must make \text{Grasped}(\text{x}0: \text{robot}, \text{x}1: \text{dot}) true for a particular set of dots that varies per task.

**Satellites Environment Details**

- **Types**:
  - The satellite type has features $x$, $y$, theta, instrument, calibration_obj_id, is_calibrated, read_obj_id, shoots_chem_x, shoots_chem_y.
  - The object type has features id, $x$, $y$, has_chem_x, has_chem_y.
- **Action space**: $\mathbb{R}^{10}$. Actions are 10-dimensional vectors: [satellite_x, satellite_y, object_x, object_y, target_satellite_x, target_satellite_y, shoot_something_x, shoot_something_y, use_instrument, calibration], which represent current satellite displacement, object displacement, target satellite location, values representing Chemical_X, Chemical_Y, the current instrument (camera (0.0 - 0.33), infrared (0.33 - 0.66), Geiger (0.66 - 1.0)), and current calibration, respectively.
  - **Predicates**:
    - \text{Sees}(\text{x}0: \text{satellite}, \text{x}1: \text{object}), \text{CalibrationTarget}(\text{x}0: \text{satellite}, \text{x}1: \text{object}), \text{IsCalibrated}(\text{x}0: \text{satellite}), \text{HasCamera}(\text{x}0: \text{satellite}), \text{HasInfrared}(\text{x}0: \text{satellite}), \text{HasGeiger}(\text{x}0: \text{satellite}), \text{ShootsChemX}(\text{x}0: \text{satellite}), \text{ShootsChemY}(\text{x}0: \text{satellite}), \text{HasChemX}(\text{x}0: \text{satellite}), \text{HasChemY}(\text{x}0: \text{satellite}), \text{CameraReadingTaken}(\text{x}0: \text{satellite}, \text{x}1: \text{object}), \text{InfraredReadingTaken}(\text{x}0: \text{satellite}, \text{x}1: \text{object}), \text{GeigerReadingTaken}(\text{x}0: \text{satellite}, \text{x}1: \text{object}).
  - **Controllers**:
    - \text{MoveTo}(\text{x}0: \text{satellite}, \text{x}1: \text{object}, \text{x}, \text{y}): Moves the satellite $\text{x}0$ to be at $x$, $y$.
    - \text{Calibrate}(\text{x}0: \text{satellite}, \text{x}1: \text{object}): Tries to calibrate the satellite $\text{x}0$ against object $\text{x}1$. This will only succeed (i.e., make \text{IsCalibrated}(\text{x}0) true) if $\text{x}1$ is the calibration target of $\text{x}0$.
    - \text{ShootChemX}(\text{x}0: \text{satellite}, \text{x}1: \text{object}): Tries to shoot a pellet of chemical X from satellite $\text{x}0$. This will only succeed if $\text{x}0$ both has chemical X and is capable of shooting it.
    - \text{ShootChemY}(\text{x}0: \text{satellite}, \text{x}1: \text{object}): Tries to shoot a pellet of chemical Y from satellite $\text{x}0$. This will only succeed if $\text{x}0$ both has chemical Y and is capable of shooting it.
    - \text{UseInstrument}(\text{x}0: \text{satellite}, \text{x}1: \text{object}): Tries to use the instrument possessed by $\text{x}0$ on object $\text{x}1$ (note that we assume $\text{x}0$ only possesses a single instrument).
  - **Goal**: The agent must take particular readings (i.e some combination of CameraReadingTaken(\text{x}0: \text{satellite}, \text{x}1: \text{object}), InfraredReadingTaken(\text{x}0: \text{satellite}, \text{x}1: \text{object}), GeigerReadingTaken(\text{x}0: \text{satellite}, \text{x}1: \text{object})).
Painting Environment Details

- **Types:**
  - The object type has features \( x, y, z, \text{dirtiness}, \text{wetness}, \text{color}, \text{grasp}, \text{held} \).
  - The box type has features \( x, y, \text{color} \).
  - The lid type has features \( x, y, \text{color} \).
  - The robot type has features \( x, y, \text{fingers} \).

- **Action space:** \( \mathbb{R}^7 + \{0, 1\} \). Actions are 8-dimensional vectors: \([x, y, z, \text{grasp}, \text{water level}, \text{heat level}, \text{color}, \text{pickplace}] \). Note that pickplace is 1 for pick, -1 for place, and 0 otherwise, while grasp, water level, heat level, and color are in \([0, 1]\).

- **Predicates:**
  - \( \text{InBox}(x_0: \text{obj}) \), \( \text{InShelf}(x_0: \text{obj}) \), \( \text{IsBoxColor}(x_0: \text{obj}, x_1: \text{box}) \), \( \text{IsShelfColor}(x_0: \text{obj}, x_1: \text{shelf}) \), \( \text{GripperOpen}(x_0: \text{robot}) \), \( \text{HoldingTop}(x_0: \text{obj}) \), \( \text{OnTable}(x_0: \text{obj}) \), \( \text{NotOnTable}(x_0: \text{obj}) \), \( \text{IsWet}(x_0: \text{obj}) \), \( \text{IsDry}(x_0: \text{obj}) \), \( \text{IsDirty}(x_0: \text{obj}) \), \( \text{IsClean}(x_0: \text{obj}) \).

- **Controllers:**
  - \( \text{Pick}(x_0: \text{robot}, x_1: \text{obj}, \text{[grasp]}) \): picks up a particular object, if grasp \( > 0.5 \) it performs a top grasp otherwise a side grasp.
  - \( \text{Wash}(x_0: \text{robot}) \): washes the object in hand, which is needed to clean the object.
  - \( \text{Dry}(x_0: \text{robot}) \): dries the object in hand, which is needed after you wash the object.
  - \( \text{Paint}(x_0: \text{robot}, \text{[color]}) \): paints the object in hand a particular color specified by the continuous parameter.
  - \( \text{Place}(x_0: \text{robot}, x_1: \text{obj}, x, y, z) \): places the object in hand at a particular \( x, y, z \) location specified by the continuous parameters.
  - \( \text{OpenLid}(x_0: \text{robot}, x_1: \text{lid}) \): opens a specific lid, which is needed to place objects inside the box.
  - \( \text{MoveToObj}(x_0: \text{robot}, x_1: \text{obj}, x) \): moves to a particular object with certain displacement \( x \).
  - \( \text{MoveToBox}(x_0: \text{robot}, x_1: \text{box}, x) \): moves to a particular box with certain displacement \( x \).
  - \( \text{MoveToShelf}(x_0: \text{robot}, x_1: \text{shelf}, x) \): moves to a particular shelf with certain displacement \( x \).

- **Goal:** A robot in 3D must pick, wash, dry, paint, and then place various objects in order to get \( \text{InBox}(x_0: \text{obj}) \) and \( \text{IsBoxColor}(x_0: \text{obj}, x_1: \text{box}) \), or \( \text{InShelf}(x_0: \text{obj}) \) and \( \text{IsShelfColor}(x_0: \text{obj}, x_1: \text{shelf}) \) true for particular goal objects.

RNT Painting Environment Details

- **Types:**
  - The object type has features \( x, y, z, \text{dirtiness}, \text{wetness}, \text{color}, \text{grasp}, \text{held} \).
  - The box type has features \( x, y, \text{color} \).
  - The lid type has features \( x, y, \text{color} \).
  - The shelf type has features \( x, y, \text{color} \).

- **Goal:** A robot in 3D must pick, wash, dry, paint, and then place various objects in order to get \( \text{InBox}(x_0: \text{obj}) \) and \( \text{IsBoxColor}(x_0: \text{obj}, x_1: \text{box}) \), or \( \text{InShelf}(x_0: \text{obj}) \) and \( \text{IsShelfColor}(x_0: \text{obj}, x_1: \text{shelf}) \) true for particular goal objects. In contrast to the previous painting environment, we also need to navigate to the right objects (i.e. all objects are not always reachable from any states). This version of the environment requires operators with ignore effects.
Table 1: Ours vs. operator-learning baselines. Percentage of evaluation tasks solves, average number of nodes expanded by task planning, and average learning time for all of our operator-learning baselines across all our environments. Results are averaged across 10 random seeds.

| Environment          | Ours | Cluster and Intersect | LOFT | Pred. Error |
|----------------------|------|-----------------------|------|-------------|
|                      | Succ | Node | Learn Time | Succ | Node | Learn Time | Succ | Node | Learn Time |
| Screws               | 100.0 | 117.3 | 0.6       | 0.0  | –    | 0.2        | 0.0  | –    | 143.7      | 50.0 | 135.1 | 715.4      |
| Repeated NextTo      | 100.0 | 88.5  | 23.9      | 8.6  | 1486  | 66.2       | 8.6  | 1348 | 67.1        | 86.2 | 44.9  | 29.7        |
| RNT Painting         | 99.0  | 1831  | 99.1      | 0.0  | –    | 486.4      | 0.0  | –    | 1718.9      | 0.0  | –     | 2897.2     |
| Painting             | 98.8  | 2749  | 69.4      | 99.0 | 1727  | 86.1       | 0.0  | –    | 110.3       | 93.4 | 1480  | 82.6        |
| Satellites           | 94.6  | 6668  | 68.1      | 0.6  | –    | 3310       | 0.0  | –    | 346.9       | 2.4  | 7546  | 89.1        |
| Satellites Simple    | 98.4  | 278.2 | 38.7      | 90.4 | 1523  | 30.1       | 0.0  | –    | 58.3        | 95.6 | 989.7 | 21.9        |

A.7 Additional Approach Details

Here we provide detailed descriptions of each approach evaluated in experiments. See Section 5.1 for high-level descriptions and the accompanying code for implementations.

Backchaining: Our main approach, operator learning via iterative backchaining.

- **Operator Learning**: Operators are learned via running our Backchaining Algorithm. The algorithm starts with the first trajectory in the dataset and iterates through the remaining trajectories in the same order as provided.

- **Sampler Learning**: Following Chitnis et al. (2022), each sampler consists of two neural networks: a generator and a discriminator. The generator outputs the mean and diagonal covariance of a Gaussian, using an exponential linear unit (ELU) to assure PSD covariance. The generator is a fully-connected neural network with two hidden layers of size 32, trained with Adam for 50,000 epochs with a learning rate of 1e−3 using Gaussian negative log likelihood loss. The discriminator is a binary classifier of samples output by the generator. Negative examples for the discriminator are collected from other skill datasets. The classifier is a fully-connected neural network with two hidden layers of size 32, trained with Adam for 10,000 epochs with a learning rate of 1e−3 using binary cross entropy loss. During planning, the generator is rejection sampled using the discriminator for up to 100 tries, after which the last sample is returned.

- **Planning**: The number of abstract plans \( N_{\text{abstract}} = 8 \) and samples per step \( N_{\text{samples}} = 10 \) for all environments. These smaller numbers were selected for the sake of experiments finishing faster. To generate plans we used A search with LMCut heuristic.

Cluster and Intersect: This is the operator learning approach used by Silver et al. (2022).

- **Operator Learning**: This approach learns STRIPS operators by attempting to induce a different operator for every set of unique lifted effects (See Silver et al. (2022) for more information).

- **Sampler Learning and Planning**: Same as Backchaining (See Section A.7 for more details).

LOFT: This is the operator learning approach used by Silver et al. (2021b).

- **Operator Learning**: This approach learns STRIPS operators similar to the Cluster and Intersect baseline, except that it uses search to see if it can modify the operators after performing Cluster and Intersect (See Silver et al. (2021b) for more information).

- **Sampler Learning and Planning**: Same as Backchaining (See Section A.7 for more details).

Prediction Error: A baseline variant of LOFT that is capable of learning operators with ignore effects.

- **Operator Learning**: This approach first runs Cluster and Intersect, then attempts to induce ignore effects by performing a hill-climbing search over possible choices of ignore effects using prediction error as the metric to be optimized.

- **Sampler Learning and Planning**: Same as Backchaining (See Section A.7 for more details).

GNN Shooting: This approach trains a graph neural network (GNN) (Battaglia et al. 2018) policy. This GNN takes in the current state \( s \), abstract state \( s = \text{ABSTRACT}(x, \Psi_G) \), and goal \( g \). It outputs an action via a one-hot vector over \( \mathcal{C} \) corresponding to which controller to execute, one-hot vectors over all objects at each discrete argument position, and a vector of continuous arguments. We train the GNN using behavior cloning on the dataset \( \mathcal{D} \). At evaluation time, we sample trajectories by treating the GNN’s output continuous arguments as the mean of a Gaussian with fixed variance. We use the known transition model \( f \) to check if the goal is achieved, and repeat until the planning timeout is reached.

- **Planning**: Repeat until the goal is reached: query the model on the current state, abstract state, and goal to get a ground skill. Invoke the ground skill’s sampler up to 100 times to find a subgoal that leads to the abstract successor state predicted by the skill’s operator. If successful, simulate the state forward; otherwise, terminate with failure.

- **Learning**: This approach learns a TAMP planner in the form of a GNN. Following the baselines presented in prior work (Chitnis et al. 2022), the GNN is a standard encode-process-decode architecture with 3 message passing steps. Node and edge modules are fully-connected neural networks with two hidden layers of size 32.
16. We follow the method of Chitnis et al. (2022) for encoding object-centric states, abstract states, and goals into graph inputs. To get graph outputs, we use node features to identify the object arguments for the skill and a global node with a one-hot vector to identify the skill identity. The models are trained with Adam for 1000 epochs with a learning rate of $10^{-3}$ and batch size 128 using MSE loss.

**GNN Model-Free:** A baseline that uses the same trained GNN as above, but at evaluation time, directly executes the policy instead of checking execution using $f$. This has the advantage of being more efficient during evaluation than GNN Shooting, but is less effective.

### A.8 Additional Experimental Results

Figure 1 shows the success rate %, average number of nodes expanded during task planning, and average wall-clock learning time for each of our environments and methods trained on 50 data points. We can see that our approach achieves the highest success rate across all environments. Moreover, it achieves the lowest number of nodes expanded in 3 out of 6 environments. The methods that require a smaller number of nodes expanded in the remaining 3 environments generally have a much lower success rate, except for Cluster and Intersect on the Painting environment, which achieves comparable success rate with fewer nodes expanded. This evidence further shows that our approach is able to learn operators that balance the effectiveness and efficiency of planning across a wide-variety of domains (Q1). Additionally, our approach has the lowest learning time in 4 out of the 6 environments. In the other 2 environments, its learning time is comparable to the best approach, and its success rate is generally higher.

### A.9 Learned Operator Examples

See Figures 6, 5, 7, 8, 9, 10, and 11 for the full set of operators learned by our algorithm in the Screws, Repeated NextTo, RNT Painting, and Satellites environments. Some operators to note include Op7, Op8, Op9, and Op10 in Figures 8 and 9 where our algorithm is able to learn two different operators corresponding to picking with a side grasp and top grasp that are linked to the same Pick controller. This is beneficial in this environment, since reliably placing into a box and shelf require holding an object with a side grasp and top grasp respectively. Additionally, note that Op1 in Figure 10 correctly discerns that Sees is an ignore effect of moving a satellite.
Op0:
Parameters: [?x0:robot, ?x1:dot]
 Preconditions: [NextTo(?x0:robot, ?x1:dot)]
 Add Effects: [Grasped(?x0:robot, ?x1:dot)]
 Delete Effects: [NextTo(?x0:robot, ?x1:dot)]
 Ignore Effects: [NextToNothing]
Controller: MoveGrasp(?x0:robot, ?x1:dot)

Op1:
Parameters: [?x0:robot, ?x1:dot]
 Preconditions: []
 Add Effects: [NextTo(?x0:robot, ?x1:dot)]
 Delete Effects: [NextToNothing(?x0:robot)]
 Ignore Effects: [NextTo]
Controller: MoveGrasp(?x0:robot, ?x1:dot)

Figure 5: RepeatedNextTo learned operators.
Op0:
    Parameters: [?x0:gripper, ?x1:receptacle, ?x2:screw]
    Preconditions: [AboveReceptacle(?x0:gripper, ?x1:receptacle), Pickable(?x0:gripper, ?x2:screw), HoldingScrew(?x0:gripper, ?x2:screw)]
    Add Effects: [ScrewInReceptacle(?x2:screw, ?x1:receptacle)]
    Delete Effects: [HoldingScrew(?x0:gripper, ?x2:screw)]
    Ignore Effects: [HoldingScrew, ScrewInReceptacle]
    Controller: DemagnetizeGripper(?x0:gripper)

Op1:
    Parameters: [?x0:gripper, ?x1:screw]
    Preconditions: [Pickable(?x0:gripper, ?x1:screw)]
    Add Effects: [HoldingScrew(?x0:gripper, ?x1:screw)]
    Delete Effects: []
    Ignore Effects: [HoldingScrew]
    Controller: MagnetizeGripper(?x0:gripper)

Op2:
    Parameters: [?x0:gripper, ?x1:receptacle, ?x2:screw]
    Preconditions: [Pickable(?x0:gripper, ?x2:screw), HoldingScrew(?x0:gripper, ?x2:screw)]
    Add Effects: [AboveReceptacle(?x0:gripper, ?x1:receptacle)]
    Delete Effects: []
    Ignore Effects: []
    Controller: MoveToReceptacle(?x0:gripper, ?x1:receptacle, ?x2:screw)

Op3:
    Parameters: [?x0:gripper, ?x1:screw]
    Preconditions: []
    Add Effects: [Pickable(?x0:gripper, ?x1:screw)]
    Delete Effects: []
    Ignore Effects: [Pickable]
    Controller: MoveToScrew(?x0:gripper, ?x1:screw)

Figure 6: Screws learned operators.
Op0:
    Parameters: [?x0:robot, ?x1:obj]
    Preconditions: [Holding(?x1:obj), IsClean(?x1:obj), IsWet(?x1:obj), NextTo(?x0:robot, ?x1:obj)]
    Add Effects: [IsDry(?x1:obj)]
    Delete Effects: [IsWet(?x1:obj)]
    Ignore Effects: []
    Controller: Dry(?x0:robot)

Op1:
    Parameters: [?x0:robot, ?x1:box, ?x2:obj]
    Preconditions: [Holding(?x2:obj), HoldingTop(?x2:obj), NextTo(?x0:robot, ?x2:obj), NextToTable(?x0:robot)]
    Add Effects: [NextTo(?x0:robot, ?x2:obj), NextToBox(?x0:robot, ?x1:box), NotOnTable(?x2:obj)]
    Delete Effects: [NextToTable(?x0:robot), OnTable(?x2:obj)]
    Ignore Effects: [NextTo]
    Controller: MoveToBox(?x0:robot, ?x1:box)

Op2:
    Parameters: [?x0:robot, ?x1:obj]
    Preconditions: [GripperOpen(?x0:robot), OnTable(?x1:obj), NextToTable(?x0:robot)]
    Add Effects: [NextTo(?x0:robot, ?x1:obj), NextToTable(?x0:robot)]
    Delete Effects: []
    Ignore Effects: [NextTo, NextToBox, NextToShelf]
    Controller: MoveToObj(?x0:robot, ?x1:obj)

Op3:
    Parameters: [?x0:robot, ?x1:shelf, ?x2:obj]
    Preconditions: [Holding(?x2:obj), HoldingSide(?x2:obj), NextTo(?x0:robot, ?x2:obj), NextToTable(?x0:robot)]
    Add Effects: [NextTo(?x0:robot, ?x2:obj), NextToShelf(?x0:robot, ?x1:shelf), NotOnTable(?x2:obj)]
    Delete Effects: [NextToTable(?x0:robot), OnTable(?x2:obj)]
    Ignore Effects: [NextTo]
    Controller: MoveToShelf(?x0:robot, ?x1:shelf)

Figure 7: RNT Painting learned operators (1/3).
Op4:
  Parameters: [?x0:robot, ?x1:lid]
  Preconditions: [GripperOpen(?x0:robot)]
  Add Effects: []
  Delete Effects: []
  Ignore Effects: []
  Controller: OpenLid(?x0:robot, ?x1:lid)

Op5:
  Parameters: [?x0:robot, ?x1:obj, ?x2:shelf]
  Preconditions: [Holding(?x1:obj), IsClean(?x1:obj), IsDry(?x1:obj), NextTo(?x0:robot, ?x1:obj)]
  Add Effects: [IsShelfColor(?x1:obj, ?x2:shelf)]
  Delete Effects: []
  Ignore Effects: []
  Controller: Paint(?x0:robot)

Op6:
  Parameters: [?x0:robot, ?x1:obj, ?x2:box]
  Preconditions: [Holding(?x1:obj), HoldingTop(?x1:obj), IsClean(?x1:obj), IsDry(?x1:obj), NextTo(?x0:robot, ?x1:obj)]
  Add Effects: [IsBoxColor(?x1:obj, ?x2:box)]
  Delete Effects: []
  Ignore Effects: []
  Controller: Paint(?x0:robot)

Op7:
  Parameters: [?x0:robot, ?x1:obj]
  Preconditions: [GripperOpen(?x0:robot), NextTo(?x0:robot, ?x1:obj), NextToTable(?x0:robot), OnTable(?x1:obj)]
  Add Effects: [Holding(?x1:obj), HoldingSide(?x1:obj)]
  Delete Effects: [GripperOpen(?x0:robot)]
  Ignore Effects: []
  Controller: Pick(?x0:robot, ?x1:obj)

Op8:
  Parameters: [?x0:robot, ?x1:obj]
  Preconditions: [GripperOpen(?x0:robot), NextTo(?x0:robot, ?x1:obj), NextToTable(?x0:robot), OnTable(?x1:obj)]
  Add Effects: [Holding(?x1:obj), HoldingTop(?x1:obj)]
  Delete Effects: [GripperOpen(?x0:robot)]
  Ignore Effects: []
  Controller: Pick(?x0:robot, ?x1:obj)

Figure 8: RNT Painting learned operators (2/3).
Op9:
    Parameters: [?x0:robot, ?x1:obj, ?x2:box]
    Preconditions: [Holding(?x1:obj), HoldingTop(?x1:obj), IsBoxColor(?x1:obj, ?x2:box), IsClean(?x1:obj), IsDry(?x1:obj), NextTo(?x0:robot, ?x1:obj), NextToBox(?x0:robot, ?x2:box), NotOnTable(?x1:obj)]
    Add Effects: [GripperOpen(?x0:robot), InBox(?x1:obj, ?x2:box)]
    Delete Effects: [Holding(?x1:obj), HoldingTop(?x1:obj)]
    Ignore Effects: []
    Controller: Place(?x0:robot)

Op10:
    Parameters: [?x0:robot, ?x1:obj, ?x2:shelf]
    Preconditions: [Holding(?x1:obj), HoldingSide(?x1:obj), IsClean(?x1:obj), IsDry(?x1:obj), IsShelfColor(?x1:obj, ?x2:shelf), NextTo(?x0:robot, ?x1:obj), NextToShelf(?x0:robot, ?x2:shelf), NotOnTable(?x1:obj)]
    Add Effects: [GripperOpen(?x0:robot), InShelf(?x1:obj, ?x2:shelf)]
    Delete Effects: [Holding(?x1:obj), HoldingSide(?x1:obj)]
    Ignore Effects: []
    Controller: Place(?x0:robot)

Op11:
    Parameters: [?x0:robot, ?x1:obj]
    Preconditions: [Holding(?x1:obj), NextTo(?x0:robot, ?x1:obj), NextToTable(?x0:robot), NotOnTable(?x1:obj)]
    Add Effects: [GripperOpen(?x0:robot), OnTable(?x1:obj)]
    Delete Effects: [Holding(?x1:obj), HoldingSide(?x1:obj), HoldingTop(?x1:obj), NotOnTable(?x1:obj)]
    Ignore Effects: []
    Controller: Place(?x0:robot)

Op12:
    Parameters: [?x0:robot, ?x1:obj]
    Preconditions: [Holding(?x1:obj), IsDirty(?x1:obj), IsDry(?x1:obj), NextTo(?x0:robot, ?x1:obj)]
    Add Effects: [IsClean(?x1:obj), IsWet(?x1:obj)]
    Delete Effects: [IsDirty(?x1:obj), IsDry(?x1:obj)]
    Ignore Effects: []
    Controller: Wash(?x0:robot)

Figure 9: RNT Painting learned operators (3/3).
Op0:
Parameters: [\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\)]
Preconditions: [CalibrationTarget(\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\)), Sees(\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\))]  
Add Effects: [IsCalibrated(\(\text{x0}:\text{satellite}\))]  
Delete Effects: []  
Ignore Effects: []  
Controller: Calibrate(\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\))

Op1:
Parameters: [\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\)]
Preconditions: []  
Add Effects: [Sees(\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\))]  
Delete Effects: []  
Ignore Effects: [Sees]  
Controller: MoveTo(\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\))

Op2:
Parameters: [\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\)]
Preconditions: [Sees(\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\)), ShootsChemX(\(\text{x0}:\text{satellite}\))]  
Add Effects: [HasChemX(\(\text{x1}:\text{object}\))]  
Delete Effects: []  
Ignore Effects: []  
Controller: ShootChemX(\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\))

Op3:
Parameters: [\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\)]
Preconditions: [Sees(\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\)), ShootsChemY(\(\text{x0}:\text{satellite}\))]  
Add Effects: [HasChemY(\(\text{x1}:\text{object}\))]  
Delete Effects: []  
Ignore Effects: []  
Controller: ShootChemY(\(\text{x0}:\text{satellite}, \text{x1}:\text{object}\))
Op4:
   Parameters: [?x0:satellite, ?x1:object]
   Preconditions: [HasCamera(?x0:satellite), 
   HasChemX(?x1:object), IsCalibrated(?x0:satellite), 
   Sees(?x0:satellite, ?x1:object)]
   Add Effects: [CameraReadingTaken(?x0:satellite, 
   ?x1:object)]
   Delete Effects: []
   Ignore Effects: []
   Controller: UseInstrument(?x0:satellite, ?x1:object)
Op5:
   Parameters: [?x0:satellite, ?x1:object]
   Preconditions: [HasGeiger(?x0:satellite), 
   IsCalibrated(?x0:satellite), Sees(?x0:satellite, 
   ?x1:object)]
   Add Effects: [GeigerReadingTaken(?x0:satellite, 
   ?x1:object)]
   Delete Effects: []
   Ignore Effects: []
   Controller: UseInstrument(?x0:satellite, ?x1:object)
Op6:
   Parameters: [?x0:satellite, ?x1:object]
   Preconditions: [HasChemY(?x1:object), 
   HasInfrared(?x0:satellite), IsCalibrated(?x0:satellite), 
   Sees(?x0:satellite, ?x1:object)]
   Add Effects: [InfraredReadingTaken(?x0:satellite, 
   ?x1:object)]
   Delete Effects: []
   Ignore Effects: []
   Controller: UseInstrument(?x0:satellite, ?x1:object)

Figure 11: Satellites learned operators (2/2).