Learning Skill Hierarchies from Predicate Descriptions and Self-Supervision

Tom Silver*, Rohan Chitnis*, Anurag Ajay, Josh Tenenbaum, Leslie Pack Kaelbling
Massachusetts Institute of Technology
{tslvr, ronuchit, aajay, jbt, lpk}@mit.edu

Abstract

We consider the problem of learning *skills* — lifted, goal-conditioned policies and associated STRIPS operators — for deterministic domains where states are represented as sets of fluents. The agent is equipped with a STRIPS planner and a set of *primitive actions*, but is not given models of these actions to use for planning. Its objective is to learn a set of policies and operators with which it can efficiently solve a variety of tasks presented only at test time. Previous works have examined the problems of learning operators and learning hierarchical, compositional policies in isolation; our focus here is to have one agent learn both. We approach this problem in two phases. First, we use inductive logic programming to learn *primitive operators* — preconditions and effects for each primitive action — from interactions with the world. Next, we use self-supervision to learn both higher-level *lifted policies* built on these primitives and their associated *operators*. We demonstrate the utility of our approach in two domains: *Rearrangement* and *Minecraft*. We evaluate the extent to which our learned policies generalize and compose to solve new, harder tasks at test time. Our work illustrates that a rich, structured library of skills can be derived from limited interactions with a predicate-based environment.

Introduction

An intelligent agent must have skills that *generalize* and *compose*. The former property demands that the agent's skills are useful in a variety of settings, which may be significantly different from the settings in which these skills were acquired. The latter allows the skills to be sequenced to solve novel, complex tasks, resulting in a *beneficial* combinatorial explosion of planning problems that can be solved.

For instance, consider a skill for a household robot that moves an object from one room in the house to another. Ideally, this skill should generalize across objects: regardless of whether it is a laptop, spoon, or book, the agent must move to it, grasp it, transport it to the target room, and place it. Furthermore, this skill should seamlessly compose with other skills, such as cleaning the object or using it as a tool.

In this work, we address the problem of learning a rich library of generalizable and composable skills from a limited number of interactions with the environment. We formalize a skill as 1) a *policy* and 2) an associated STRIPS *operator description*, which captures the preconditions of this policy and the effects of executing it in the world.

To facilitate generalization, we seek to learn policies that are *lifted* and *goal-conditioned*. The policies are lifted in the sense that they are parameterized, e.g. `Transport(robot-location, object, target-room)`, which should work over a variety of objects and rooms that could be passed in as arguments. The policies are goal-conditioned in that they are parameterized by not only variables describing the current state (`robot-location`), but also variables describing the policy's goal (`object` and `target-room`). The STRIPS operators associated with each policy are compositional by design; classical planners (Hoffmann 2001; Helmert 2006) can

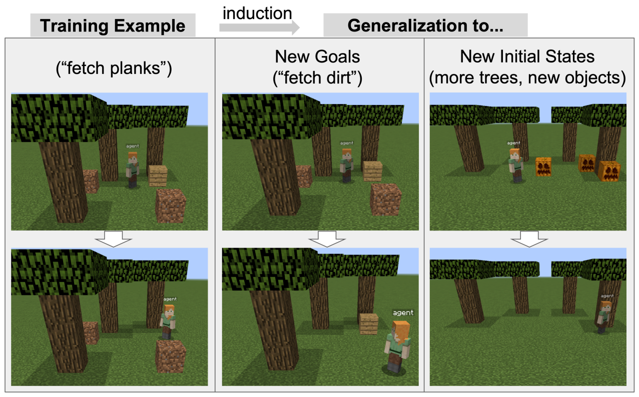


Figure 1: We learn lifted, goal-conditioned policies and associated STRIPS operators for predicate-based domains. The policies we learn can generalize to both new goals and new initial states containing novel object types and arbitrary numbers of objects. Pictured is our *Minecraft* domain, where after examples of performing a handful of tasks such as fetching planks, our agent can accomplish new goals (fetch two dirts in sequence) and generalize to new initial states, which may include new object types such as pumpkins or more trees (see top-right image).

* Equal contribution.

efficiently sequence them to solve long-horizon tasks.

We assume a deterministic domain where states are represented as sets of fluents and the agent is given a set of primitive parameterized actions such as `Move`, `Pick`, and `Place`. We then proceed in two phases. First, we use inductive logic programming (Blockeel and De Raedt 1998; Lavrac and Dzeroski 1994; Muggleton 1991) to learn primitive operators — preconditions and effects for the primitive actions — from interactions with the world. Next, we employ a STRIPS planner over the primitive operators to simulate new environment interactions (within the agent's "mind"), from which we synthesize skills: higher-level lifted policies built on the primitives and their associated operators. The operators are synthesized using backward chaining (goal regression) (Kaelbling and Lozano-Pérez 2010; Pollock 1998; Alcázar et al. 2013). This skill synthesis step is run repeatedly to build increasingly robust policies and correct associated operators. With these operators, a planner can chain together the policies to solve more complex future tasks.

We demonstrate the utility of our approach in two discrete, deterministic domains: *Rearrangement* and *Minecraft*. Our experiments are designed to show both generalizability and compositionality of our learned policies and operators. To show generalizability, we evaluate policies when 1) starting in initial states vastly different from those seen during training and 2) they are given arguments not seen during training. To show compositionality, we evaluate whether a STRIPS planner can use our learned operators to find plans for goals more complex than those seen during training.

Background

STRIPS Planning

A STRIPS (Fikes and Nilsson 1971) planning domain is a tuple $\langle \mathcal{P}, \mathcal{O} \rangle$, where $\mathcal{P}$ is a set of predicates (Boolean-valued functions) and $\mathcal{O}$ is a set of operators consisting of: a set of discrete parameters, a set of preconditions specifying when that operator can be executed, and a set of effects that hold after the operator is executed.

A STRIPS planning problem instance is a tuple $\langle I, G \rangle$, where I is an initial state that contains 1) a set of objects $\mathcal{X}$ in the domain and 2) a set of fluents that hold initially. A fluent is the application of a predicate to objects in $\mathcal{X}$. The goal G is described as a conjunction of fluents.

We will often use the phrase "planning problem" to refer to a STRIPS domain and problem together. The solution to a planning problem is an open-loop plan — a sequence of operators $a_1, ..., a_n \in \mathcal{A}$ with corresponding arguments chosen from $\mathcal{X}$ — such that starting from the fluents in I and applying the operators' effects in sequence, each state satisfies the preconditions of the subsequent operator and G is a subset of the final state. STRIPS planning problems are deterministic, fully observable, and discrete. The challenge in solving them comes from the combinatorial nature of the search for a plan that drives the initial state to the goal.

Inductive Logic Programming

Inductive logic programming (ILP) (Lavrac and Dzeroski 1994; Muggleton 1991) is a class of techniques for learning a Prolog hypothesis h to explain examples $\mathcal{E}$. *Learning from interpretations* (De Raedt 1996; Blockeel and De Raedt 1998) is an ILP setting in which each example is a pair of 1) a conjunction of "input" fluents Z and 2) a conjunction of "target" fluents Y.[1] We assume that the input-output mapping is a deterministic function, and therefore consider all (Z, Y') with $Y' \neq Y$ to be negative examples. A hypothesis h is thus valid if for all $(Z, Y) \in \mathcal{E}$, $Z \wedge h \implies Y$ and for all $Y' \neq Y$, $Z \wedge h \not\Longrightarrow Y'$.

Related Work

Learning STRIPS operators. Significant attention has been given recently to learning STRIPS action models from interaction data. For instance, Mourao et al. (2012) develop a method that is robust to noise in the interaction data by first learning noiseless implicit action models, then synthesizing STRIPS models from these. In contrast, we focus on leveraging induction to obtain significant generalizability, allowing our learned policies to solve problem instances very different from those seen in the interaction data used for training. To make our methods robust to noise, one can turn to probabilistic ILP systems (De Raedt and Kersting 2008).

Fikes, Hart, and Nilsson (1972) propose an approach for generalizing plans found by a STRIPS planner, via replacing certain ground terms in the plan steps with problem-independent variables, to create so-called *macrops*. Minton (1985) extends this work to address the problem of overwhelming macrop proliferation, selectively choosing only certain plans to generalize using a heuristic. Like macrop learning, our method achieves a similar effect in learning lifted policies, but we do not assume that the agent starts off with any operators. Wang (1996) learns STRIPS operators from solution traces of an expert performing a task, while Wang et al. (2018) learn geometric constraints on the parameters of operators, representing part of the required preconditions. Unlike these approaches, we learn complete STRIPS action models from the agent's own interaction data.

Generalizable options for reinforcement learning. Options provide a formalism for specifying temporally extended actions within the reinforcement learning framework. They consistent of a policy, an initiation set, and a termination condition, similar to the policies and operator preconditions that we learn. Particularly relevant are works where a set of options is learned so that the agent can accomplish a variety of new tasks at test time (Oh et al. 2017; Tessler et al. 2017; Jinnai et al. 2019). An option *model* specifies the outcome of executing an option from an initial state, similar to the effects that we learn. Options and

[1]This description differs from that given by Blockeel and De Raedt (1998) in two ways: 1) a background theory B, when available, can be included in each Z (so that $Z' = Z \cup B$), so we do not separately define B; 2) our targets are *conjunctions* of fluents *with arguments*; this generalizes previous work in which targets are single zero-arity symbols.

their models are not typically *lifted* like the representations we study in this work. Much work has addressed the problem of learning these options (Konidaris and Barto 2009; Stolle and Precup 2002), though fewer consider learning the option models (Sutton, Precup, and Singh 1999).

Thrun and Schwartz (1995) develop a method for identifying and extracting skills, partial policies defined only over a subset of the state space, that can be used across multiple tasks. Their method selects skills that minimize both the loss in performance due to using this skill rather than the low-level actions, and the complexity of the skill description. Konidaris and Barto (2007) suggest to learn options in agent-space rather than problem-space so that they can more naturally be transferred across problem instances. A separate line of work leverages motor primitives for learning higher-level motor skills (Peters and Schaal 2008); our method can similarly be seen as a way of learning higher-level control policies on top of a given set of primitive operators.

Reinforcement learning typically requires a high number of interactions with the environment in order to learn useful policies, and so we do not pursue this family of approaches. The idea of learning generalizable and reusable options is, nevertheless, related to the high-level objective of this work.

Skill learning via goal-setting and self-play. The idea of training agents to acquire skills via self-play dates back decades, and is especially prevalent as a data collection strategy when training AI for game-playing (Tesauro 1995; Silver et al. 2016). Within single-agent settings, self-play can be described as setting goals for oneself and attempting to reach them, while learning something through this process. Such approaches have led to impressive recent successes in control tasks (Held et al. 2018; Florensa et al. 2017) and planning problems such as Sokoban (Groshev et al. 2018). While these approaches allow agents to acquire diverse skills, they typically do not involve a model-learning component as ours does, which is useful because it lets us take advantage of compositionality.

Problem Formulation

We are given a deterministic environment with fully observed states $S \in \mathcal{S}$ and *primitive actions* $a \in \mathcal{A}$. States are conjunctions of fluents over a set of predicates $\mathcal{P}$, and primitive actions are literals over a different set of predicates $\mathcal{Q}$. The agent acts in the environment episodically: for each episode, an initial state S_0 and set of objects $\mathcal{X}$ are sampled from a distribution $P(I)$ over possible $I = (S_0, \mathcal{X})$, and for each action $a \in \mathcal{A}$ taken by the agent, the environment transitions to a new state following $S' = T(S, a)$. All state fluents and action literals are grounded with objects from $\mathcal{X}$. The transition function T is unknown to the agent.

The agent's first task is to learn operators for the primitive action predicates $\mathcal{Q}$. As described in "Background", an operator consists of a set of discrete parameters, a set of preconditions, and a set of effects. The operator can be grounded by passing in arguments (objects from $\mathcal{X}$) for the parameters. We use $\mathcal{O}_0$ to denote the set of primitive operators.

The agent's next mandate is to learn skills — higher-level policies building on the primitives, and associated operators — that allow it to efficiently solve a large suite of planning problems at test time. Like operators, policies are parameterized, and can be made ground by passing in objects from $\mathcal{X}$, which may encode the desired goal of the policy or information about the state. A policy π is thus a mapping from states $\mathcal{S}$ and a constant k number of objects $\mathcal{X}^k$ to primitive actions $\mathcal{A}$. Note that since $\mathcal{X}$ is a property of a problem instance, *not* the domain, these policies can naturally generalize to new object instances or types.

We use $\langle \Pi_1, \mathcal{O}_1 \rangle$ to denote a learned set of policies and operators, and $\langle \Pi, \mathcal{O} \rangle = \langle \mathcal{Q} \cup \Pi_1, \mathcal{O}_0 \cup \mathcal{O}_1 \rangle$ to denote the (unground) primitives and learned policies, together with all learned operators for both primitives and policies.

The agent is equipped with a resource-limited STRIPS planner. Given a planning problem $\langle I, G \rangle$, the agent uses its current set of operators $\mathcal{O}$ to search for a plan $\pi_1, ..., \pi_n \in \Pi$, where each step π_i may be either a ground primitive action or a ground learned policy. This plan can be executed step-by-step: for primitive actions, simply execute that action in the world, and for learned policies, continuously run the next primitive it suggests until the effects of the operator are met (or a timeout is reached). We write $\text{SUCCESS-PE}(\mathcal{O}, \Pi, I, G) \in \{0, 1\}$ to indicate whether **P**lanning followed by **E**xecution succeeds (1) or fails (0).

A good collection of skills $\langle \Pi, \mathcal{O} \rangle$ will allow the agent to solve planning problems of interest. To formalize this notion, we consider a distribution over goals $P(G)$ alongside the distribution over initial states $P(I)$, assuming independence $P(I, G) = P(I)P(G)$. The full objective is then:

$$\langle \Pi^*, \mathcal{O}^* \rangle = \underset{\Pi, \mathcal{O}}{\text{argmax}} \underset{\langle I, G \rangle}{\mathbb{E}} \left[\text{SUCCESS-PE}(\mathcal{O}, \Pi, I, G) \right].$$

This objective involves optimizing over a set of policies and operators. To make this problem more tractable, we decompose it into two separate objectives, one for operator learning and another for policy learning, which in practice can be alternated to approximately solve the full problem.

Operator Learning Objective Operators should correctly describe the preconditions and effects of their associated policies or primitive actions. Let $\pi \in \Pi$ be a policy or primitive, and $\overline{x} = (x_1, ..., x_k)$ be arguments to ground it. The objective is to learn a binary classifier $\text{Pre}(\pi, \overline{x}, S)$ and a regressor $\text{Eff}(\pi, \overline{x}, S)$. The effects regressor predicts the effects of running π with arguments $\overline{x}$, starting from state S. The precondition classifier predicts whether these effects will hold when starting from S, and must be true as much of the state and argument space as possible.

Formally, let $\delta(S, S')$ be the fluents in state S' not present in state S (positive effects) unioned with the *negation of* fluents in state S not present in state S' (negative effects). Overloading notation, let $S' = T(S, \pi)$ be the state after policy π is executed to completion starting from state S.

Given π, the objective is to learn Pre and Eff optimizing:

$$\begin{aligned} &\max \sum_{S, \overline{x}} \text{Pre}(\pi, \overline{x}, S) \\ &\text{s.t. } \forall\, S, \overline{x} : [\text{Pre}(\pi, \overline{x}, S) = 1] \implies \\ &\qquad [\text{Eff}(\pi, \overline{x}, S) = \delta(S, T(S, \pi))]. \end{aligned}$$

Policy Learning Objective Each policy should contribute to the overall objective by expanding the set of problems that can be solved with a resource-bounded planner. We approximate this objective by preferring policies that solve as many problem instances as possible *in a single step*; that is, by executing the policy from the initial state, we should arrive at the goal. Let $\text{SUCCESS-E}(\Pi, I, G) = 1$ if **E**xecuting some policy succeeds, i.e. for some $\pi \in \Pi$, we have $G \subseteq T(I, \pi)$. Maximizing this metric alone would lead us to an arbitrarily large set of policies; we must bear in mind that we ultimately want to use these policies to learn associated operators for planning, and while adding a new policy may increase the effective planning depth, it also inevitably increases the breadth. We therefore wish to avoid adding policies that are redundant or overly specialized to specific problem instances. Thus, our objective for policy learning is:

$$\Pi^* = \underset{\Pi}{\text{argmax}} \underset{\langle I,G \rangle}{\mathbb{E}} [\text{SUCCESS-E}(\Pi, I, G)] - \beta |\Pi|,$$

where $\beta > 0$ is a parameter controlling regularization. Note that we did not need this regularizer in the full objective because a resource limit was baked into our STRIPS planner, which was part of the SUCCESS-PE function evaluation.

Learning Policies and Operators

In this section, we describe our approach to learning lifted policies and STRIPS operators. An overview of the approach is provided in Figure 2.

Phase 1: Learning Primitive Operators

The first problem that we must address is learning primitive operators $\mathcal{O}_0$ — one per primitive action predicate in $\mathcal{Q}$ — from interactions with the environment. Recall that states are conjunctions of fluents and (ground) actions are ground literals. From interacting with the environment, we can collect a dataset of state-action trajectories $(S_0, a_0, S_1, ..., S_T) \in \mathcal{D}$. We use a simple strategy for data collection where the agent executes actions selected uniformly at random. More sophisticated methods for exploration and data collection could easily be incorporated to improve sample complexity.[2] After each new transition, we check to see whether the current operator for the taken action fits the transition; if so, we discard the sample and continue, and if not, we add the sample to our dataset and retrain the operator. We now describe this training procedure.

Recall that an operator O_q for an action predicate $q \in \mathcal{Q}$ consists of a precondition classifier $\text{Pre}(q, \overline{x}, S)$ and an effects regressor $\text{Eff}(q, \overline{x}, S)$. We suppose that by default, primitive actions can be executed in every state, i.e., that $\text{Pre}(q, \overline{x}, S) \equiv 1$. However, in many domains of interest, including those in our experiments, there are often action failures. For example, a `Pick` action may fail if the target object is not within reach. We accommodate this possibility by permitting a special zero-arity *Failure* predicate, the presence of which indicates that some previous action has failed. A state with *Failure* can be seen as a "sink state" for the environment, from which no action can escape. In other words, executing the `Pick` action on an unreachable target will result in a failure state, effectively ending the episode. If a *Failure* predicate exists for a domain, we will consider the preconditions of an action to include all those states that do not lead to immediate *Failure* when the respective action is taken. Formally, $\text{Pre}(q, \overline{x}, S) = 1 \iff \textit{Failure} \notin \text{Eff}(q, \overline{x}, S)$.

With the preconditions defined, the remaining problem is to learn effects. We convert the dataset of trajectories $\mathcal{D}$ into separate datasets of *effects* for each predicate q, i.e., $\mathcal{E}_q = \{(S_t, \delta(S_t, S_T))\}$ for all $(..., S_t, a_t, ..., S_T) \in \mathcal{D}$ where the predicate of a_t is q. Here, δ is the effect-retrieval function defined in "Operator Learning Objective." Now, if we can learn formulas h_q such that $S \wedge h_q \implies S'$ and $S \wedge h_q \not\Longrightarrow S''$ for all $S'' \neq S'$, then we can recover the effects we seek: $S \wedge h_q \implies \text{Eff}(q, \overline{x}, S)$. So, learning effects reduces to the inductive logic programming (ILP) problem of *learning from interpretations* (see "Background").

Among many possible ILP methods (Lavrac and Dzeroski 1994; Muggleton 1991; Quinlan 1990), we found top-down induction of first-order logical decision trees (TILDE) (Blockeel and De Raedt 1998) to be a simple and effective method for operator effect learning. TILDE is an extension of standard decision tree learning (Quinlan 1986) that allows for (unground) literals as node features. In the original formulation, TILDE considers only propositional classes; our implementation instead allows for lifted literal classes, which is necessary for the effect mapping we seek to learn. A second necessary extension is to permit multiple output literals for a single input, since there are generally multiple effects per action. While a different tree could in principle be learned for each output literal, we instead learn one tree with conjunctions of literals in the leaf nodes. Interestingly, we discover that having multiple outputs in the leaf nodes actually amounts to stronger guidance for induction and therefore makes the learning problem *easier* for TILDE.

Phase 2: Learning Lifted Policies and Operators

With primitive operators in hand, we can now proceed to learning higher-level policies and associated operators. We proceed in two phases that we iterate repeatedly. First, we use the primitive operators to learn lifted, goal-conditioned policies that invoke primitive actions. Then, we learn operators for those policies so that we can plan to use them in sequences to achieve harder goals. A natural next step, which we do not address in this work, would be to allow the policies to invoke other policies, leading to a loop where we learn policies from operators and operators from policies to grow an arbitrarily deep hierarchy of skills.

Policy Learning To optimize the policy learning objective described in "Problem Formulation," we must find a set of policies that is as small as possible while solving as many planning problems in one execution as possible. Rather than trying to optimize the size of the policy set automatically, we

[2]Interestingly, exploration strategies based on goal setting like the ones we use for policy learning fare poorly for primitive operator learning. The learned operators can be initially pessimistic about what effects can be achieved; planning with these pessimistic operators to gather more data, we will never find trajectories to correct the pessimism.

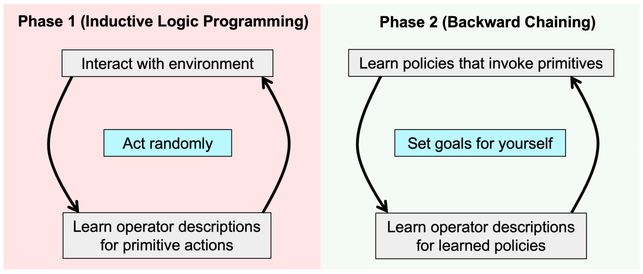

Phase 1 (Inductive Logic Programming)
Interact with environment
Act randomly
Learn operator descriptions for primitive actions
Phase 2 (Backward Chaining)
Learn policies that invoke primitives
Set goals for yourself
Learn operator descriptions for learned policies

Figure 2: An overview of our approach to learning lifted, goal-conditioned policies and STRIPS operators. In the first phase, we learn operators for the agent's primitive actions by acting randomly in the environment, and applying inductive logic programming. In the second phase, we iteratively learn higher-level policies (that invoke primitives) and associated operator descriptions by setting novel goals for the agent to achieve in its environment, and applying backward chaining.

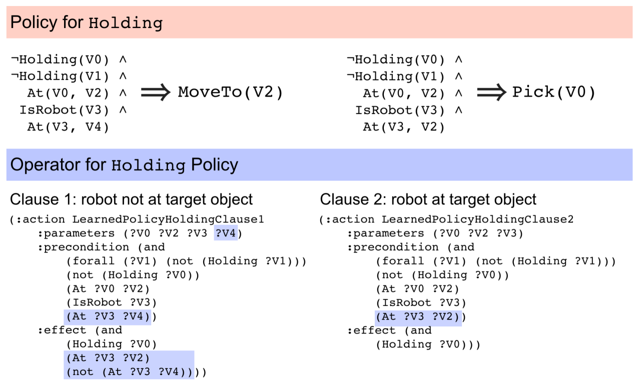
opt to decompose the learning problem into individual, independent policy learning problems, where each policy is responsible for achieving *one goal predicate*. Formally, a goal predicate is any predicate in $\mathcal{P}$ that appears in some goal $G \sim P(G)$. For example, if `Holding` is a predicate in $\mathcal{P}$, we would learn a policy that takes in a state and an object $x \in \mathcal{X}$ that the agent wants to be holding, and returns an action in furtherance of the `Holding`(x) goal. In assigning one predicate to each policy, we avoid the possibility that two learned policies will be completely redundant. We also attain good coverage of goals that the agent may encounter in future planning problems, as all goals are expressed in terms of fluents, i.e., groundings of the goal predicates.

We represent each policy as a *first-order logic decision list* (FOLDL), which is a first-order logic decision tree (Blockeel and De Raedt 1998) with linear chain structure. A FOLDL is a list of (clause, unground action) pairs $(R, q(\overline{v}))$ called *rules*, with semantics: $\forall\, \overline{x} \,.\left(\bigwedge_{j=1}^{i} R_j(\overline{x})\right) \implies q_i(\overline{x})$. We will learn FOLDL policies in such a way that the order of the rules is in correspondence with the number of steps remaining to achieve the associated goal: the first i rules will handle the case where the agent is one step away from achieving the goal; rules $i+1$ through $i+1+j$ will handle the case where the agent is two steps away; and so on. Here, i and j refer to the number of rules necessary to characterize the corresponding precondition sets.

We use a simple backward chaining strategy to learn FOLDL policies from operators. First, we describe an inefficient version of the method that relies on an exhaustive backward search. Then, we describe how to guide the backward search toward useful parts of the state space using data.

The key component of backchaining is an operation that takes an action predicate q, objects $\overline{x}$, and a state S'; and "inverts" the action, producing the set of literals from which applying q with $\overline{x}$ results in S'. We denote this mapping:

$$\mathrm{Inv}(q, \overline{x}, S') \triangleq \bigwedge \{S : \mathrm{Pre}(q, \overline{x}, S) \wedge (\mathrm{Eff}(q, \overline{x}, S) \implies S')\}.$$


Policy for Holding
Operator for Holding Policy
Clause 1: robot not at target object
Clause 2: robot at target object

```
¬Holding(V0) ∧
¬Holding(V1) ∧
  At(V0, V2) ∧   ⟹ MoveTo(V2)
 IsRobot(V3) ∧
  At(V3, V4)
```

```
¬Holding(V0) ∧
¬Holding(V1) ∧
  At(V0, V2) ∧   ⟹ Pick(V0)
 IsRobot(V3) ∧
  At(V3, V2)
```

```
(:action LearnedPolicyHoldingClause1
    :parameters (?V0 ?V2 ?V3 ?V4)
    :precondition (and
        (forall (?V1) (not (Holding ?V1)))
        (not (Holding ?V0))
        (At ?V0 ?V2)
        (IsRobot ?V3)
        (At ?V3 ?V4))
    :effect (and
        (Holding ?V0)
        (At ?V3 ?V2)
        (not (At ?V3 ?V4))))
```

```
(:action LearnedPolicyHoldingClause2
    :parameters (?V0 ?V2 ?V3)
    :precondition (and
        (forall (?V1) (not (Holding ?V1)))
        (not (Holding ?V0))
        (At ?V0 ?V2)
        (IsRobot ?V3)
        (At ?V3 ?V2))
    :effect (and
        (Holding ?V0)))
```

Figure 3: Example of a policy and associated operator learned by our approach. To be `Holding` an object, the robot must be at the same location as it; therefore, the learned policy has two clauses. The first expresses that if the robot is at some location V4 that is different from the object's location V2, then the robot should `MoveTo` V2. The second expresses that if the robot is at V2, it should `Pick` the object. (Variable uniqueness is a consequence of our TILDE implementation.)

We can compute $\mathrm{Inv}(q, \overline{x}, S')$ directly from the operator we previously learned for q, which contains explicit representations of $\mathrm{Pre}(q, \overline{x}, S)$ and $\mathrm{Eff}(q, \overline{x}, S)$. In particular, for each of the positive operator effects, we add a corresponding negative literal to $\mathrm{Inv}(q, \overline{x}, S')$; for each negative effect, we add a positive literal. In adding a positive or negative literal to $\mathrm{Inv}(q, \overline{x}, S')$, if the opposite literal already exists in the set, we cancel them out by removing both. Precondition literals are added without modification.

Suppose now that we are learning a policy for achieving groundings of goal predicate $g \in \mathcal{P}$. To do so, we begin by building a graph where nodes are sets of literals and edges are actions (both unground). The root node represents the goal predicate g applied to a set of unground variables. To build the graph inductively, we use the Inv function defined above to generate successors: for each node with literals S', create a child node for every $q \in \mathcal{Q}$ and every setting of unground variables $\overline{v}$; this child node contains literals $\mathrm{Inv}(q, \overline{v}, S')$ and is connected to its parent by an edge labeled with unground action $q(\overline{v})$.

With the graph constructed, we can read out the FOLDL representing the policy π for achieving g by traversing the tree in level order (first the root, then all depth 1 nodes, then all depth 2 nodes, etc.). For each node visited after the root, we append a new rule $(R, q(\overline{v}))$ to the FOLDL, where R is the set of literals in the node and $q(\overline{v})$ is the edge connecting the node to its parent. This procedure completes our construction of the policy.

The construction above effectively searches for all paths to the given goal g, including perhaps paths that involve rare or "dead" states that have negligible impact on our ultimate planning objective. Instead, to focus this search on higher-likelihood parts of the state space, we use our primitive operators and STRIPS planner to create a dataset of representative state-action trajectories $\tau = \langle I, q_0(\overline{v}), S_1, q_1(\overline{v}), ..., S_T \rangle$, where $S_T \implies g(\overline{v})$. (The specific means by which we acquire this dataset can greatly impact learning; see "Generating Data via Goal-Setting" below.) Given these trajectories, instead of computing the full graph, we can approximate it to only contain states seen in *some* trajectory. The complexity of policy learning is then a function of the number and length of these trajectories, rather than the size of the entire state and action space.

Operator Learning With policies learned, we must next learn their associated operators. In particular, for each policy π, we must derive a precondition classifier $\text{Pre}(\pi, \overline{x}, S)$ and an effects regressor $\text{Eff}(\pi, \overline{x}, S)$.

Observe that the FOLDL representation of π allows us to immediately read out the preconditions: any state for which the decision list implies *any* action is in the preimage of the policy. Formally, if π contains clauses $R_1, R_2, \ldots, R_k$, then:

$$\text{Pre}(\pi, \overline{x}, S) = \bigvee_{i=1}^{k} (S \implies R_i(\overline{x}))\,.$$

To learn effects, we take advantage of a backchaining procedure that is very similar to the one used for policy learning. As in policy learning, we will use FOLDLs to define the mapping for effects. However, whereas policy FOLDLs contain single literals q_i at the leaves, the FOLDLs for effects will contain conjunctions of literals E_i, as executing a policy generally results in multiple effects.

In policy learning, we constructed a graph where nodes were sets of literals and edges were actions, both unground. A path from a descendent to the root represented a trajectory leading from some initial state to a state implying the goal. For effects learning, we construct an analogous graph, using the same function Inv and starting with the same root node g applied to a set of unground variables. The key difference is that edges in this graph will now be annotated with *effects*, rather than actions. As a base case, consider the edges emanating from the root. Let S'' be the literals in the root. For a child of the root with literals S', we annotate the edge with $\delta(S', S'')$, the set of literals in S'' not in S' (positive effects) and vice versa (negative effects). Next, as an inductive step, let S be a node at depth 2 or more in the tree, and let S' be its parent. Let E' be the effects annotating the edge between S' and *its* parent. Then the effects labeling the edge between S and S' will be $E' \cup \delta(S, S')$. If two effects in this union cancel out, both are removed. As in policy learning, we can read out a FOLDL representing the effects by traversing this graph in level order, and add nodes along with their parent edges as rules to the FOLDL.

Generating Data via Goal-Setting

Now, we discuss three simple strategies for generating the data used to train the policies and associated operators. Note that the primitive operators are trained on random interaction data, but this will not be useful for training policies, since the policies must capture more interesting behavior in the world that will likely never be encountered through random interaction. We start with describing the simplest strategy; each subsequent one builds upon the one before.

Strategy 1: *Goal Babbling*. Recall that we train one policy per goal predicate in the domain; therefore, our data generator should return pairs of (τ, p) where τ represents a state-action trajectory of interaction data that turns on some grounding of predicate p. The *Goal Babbling* (Baranes and Oudeyer 2010) strategy samples problem instances (I, G), runs the planner to try to solve them, and if successful, emits the resulting state-action trajectory τ as data for turning on the predicate associated with (ground) fluent G.

Strategy 2: *Goal Babbling with Hindsight*. This strategy identifies *all* fluents that get turned on within τ, rather than just the goal G, and emits data associated with each one. To ensure that the emitted trajectories are the optimal way to achieve each fluent, which is needed so that the policies do not see inconsistent data of different action selections from the same state, the planner must be re-run for each fluent. This strategy allows the agent to glean extra information from each trajectory, about how to achieve predicates other than the one corresponding to the goal G it had set for itself.

Strategy 3: *Exhaustive Novelty Search*. This strategy exhausts all achievable goals for a particular initial state before moving on to the next one. This ensures that all goals in $P(G)$ get encountered, but it may overfit to the particular few initial states in $P(I)$ that it sees.

Training Loop

Pseudocode for the full training loop is shown in Algorithm 1. We begin by learning operators for the primitive action predicates $\mathcal{Q}$. Then, we initialize a data buffer associated with each predicate p in the domain. We iteratively train the policies on data produced by the GETDATA procedure, which can be any of the three strategies discussed previously. This procedure returns pairs of (τ, p), representing a state-action trajectory τ and the predicate p of some fluent which turns true on the final step of that trajectory.

```
Algorithm TRAIN-LOOP($\mathcal{P}, \mathcal{Q}, \mathcal{A}, P(I), P(G)$)
1   for each primitive predicate $q \in \mathcal{Q}$ do
2       Train primitive operator $O_q$, add to set $\mathcal{O}_0$.
3   for each predicate $p \in \mathcal{P}$ do
4       Initialize $\mathcal{D}_p \leftarrow$ empty data buffer.
5   while policies not converged do
6       for $\tau, p \in$ GETDATA($\mathcal{A}, \mathcal{O}_0, P(I), P(G)$) do
7           Append $\tau$ to dataset $\mathcal{D}_p$.
8           Fit policy $\pi_p$ to $\mathcal{D}_p$.
9       for each predicate $p \in \mathcal{P}$ do
10          Learn operator description $O_p$ for $\pi_p$.
```

Experiments

We conduct experiments in two domains: *Rearrangement*, implemented in the PyBullet simulator (Coumans, Bai, and Hsu 2018), and *Minecraft*, implemented on the Malmo platform (Johnson et al. 2016). See Figure 4 for visualizations. We first describe these domains, and then discuss our results.

Rearrangement Domain Description

A robot is interacting with objects of various colors located in a 3×3 grid. Its goal is expressed as a conjunction of

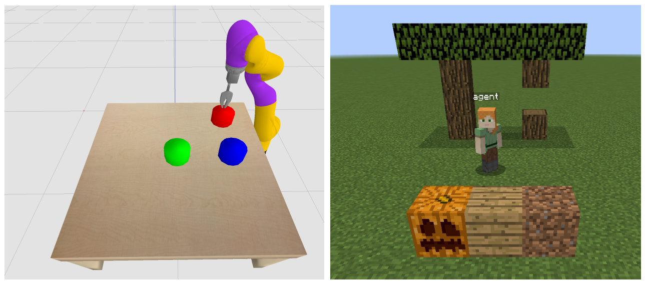


Figure 4: Visualizations of our two experimental domains. *Left:* Rearrangement, implemented in the PyBullet simulator (Coumans, Bai, and Hsu 2018). *Right:* Minecraft, implemented on the Malmo platform (Johnson et al. 2016).

(object, desired location) fluents, and thus the robot must rearrange the objects into a desired configuration. The robot's primitive actions are 1) to `MoveTo` a given grid location, 2) to `Pick` a given object from the current location (which only succeeds if the robot is not holding anything, and if it is at the same location as the object), and 3) to `Place` a given object at the current location (which only succeeds if the robot is holding that object).

We consider any predicate that can change within an episode to be a goal predicate. The two goal predicates for which we learn policies in this domain are `Holding`, parameterized by an object; and `At`, parameterized by a moveable and a location, where a moveable can be either the robot or an object. Even in this simple domain, policies can get quite complex: for instance, making an object be `At` a location requires reasoning about whether the robot is currently holding some other object, and placing it down first if so.

We design a test set of 600 problem instances in this domain, varying in difficulty from holding a single object to rearranging several objects. Many of these problems involve objects unseen during training time, as well as varying numbers of objects in the grid.

Minecraft Domain Description

This domain features an agent in a discretized Minecraft world. The agent and other objects — logs, planks, and dirt at training time — are placed in a 5×5 grid. Additionally, the agent has an inventory, which may contain arbitrarily many objects. The agent may also equip a single object from its inventory at a time. As in the *Rearrangement* domain, `MoveTo` and `Pick` are among the primitive actions. Other primitive actions include `Equip`, which allows the agent to equip an object if the object is in the inventory and another is not already equipped; `Recall`, the inverse of `Equip`; and `CraftPlanks`, which converts an equipped log into a new plank and puts it in the agent's inventory.

The goal predicates for which we learn policies in this domain are `Inventory`, parameterized by an object; `IsPlanks` and `Equipped`, each parameterized by an object; and `IsEmpty`, parameterized by a location. If an agent starts with an empty inventory and with no planks in the world, it must `MoveTo` a location with a log, `Pick` the log, `Equip` the log, and execute `CraftPlanks` in order to satisfy `IsPlanks`. Satisfying two `IsPlanks` goals would require repeating this sequence with two different logs.

We design a test set of 400 problem instances in this domain with varying difficulty in terms of the number of goal fluents and the initial states. As in *Rearrangement*, object types (e.g. novel ones such as pumpkins) and counts vary with respect to what was seen during training.

Results and Discussion

We conduct three experiments in these domains that measure the improvements yielded by each of the three major components of our approach: 1) learning primitive action operators; 2) learning lifted, goal-conditioned policies; and 3) learning operator descriptions for the learned policies.

Figure 5 shows the results of planning with learned operators for the primitive actions. These operators are learned via data of random interaction with the environment. We can see that in both domains, the quality of the operators improves over time. By inspecting the learned operators, we find that the preconditions expand to cover more of the state space and the effects become more accurate over time. As seen in the empirical results, the improved operators translate into improved test-time performance.

After learning the primitive operators, we proceed with learning lifted, goal-conditioned policies for achieving the various goal predicates in the domains. Figure 6 shows the results of invoking these policies to solve problem instances in the test set. Note that in this experiment, we have *not* learned the operator descriptions of these policies, and so we cannot yet use them in a planner; therefore, we can only achieve single-fluent goals by running the respective policy. The results show that in both domains, the policies we learn outperform invoking only the primitives after only a handful (less than 5) trajectories collected as training data.

In our final experiment, we close the loop between STRIPS planning and policies, learning operator descriptions of learned policies so that we can plan to execute multiple policies in sequence. Results of planning with these learned policies and operators are shown in Figure 7. We find that test suite performance is far improved beyond what was attainable with policies or primitive operators alone. This improved performance is an illustration of the importance of *compositional* reasoning; our operators can be composed seamlessly to achieve complex, multi-fluent goals within a limited planning horizon of 5. Most of these goals cannot be achieved by only primitive actions within that same horizon, as shown by the green curves in Figure 5. The large improvement achieved by planning with our learned operators is due to the combinatorial nature of search: each policy can "reach" farther within a single timestep than a primitive can, and so a *sequence* of these policies provides exponential increases in the attainable coverage. Therefore, many more goals can be solved, as shown by our empirical results.

Across both Figure 6 and Figure 7, we observe that the *Goal Babbling* and the *Goal Babbling with Hindsight* strategies perform equally well (with respect to the margin of error), and both slightly better than the *Exhaustive* one. This matches intuition: the *Exhaustive* strategy learns policies

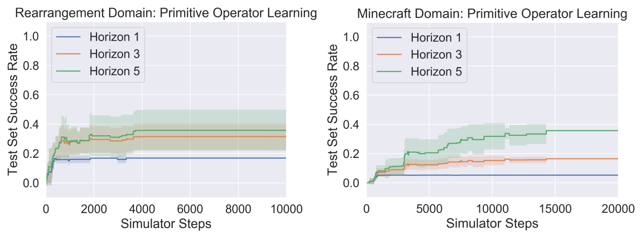


Figure 5: Test set performance versus number of interactions with environment, for STRIPS planning with learned primitive operators. In both domains, we can see that the quality of the operators improves over time. The primitives alone are insufficient to achieve perfect test performance, motivating our next experiments.

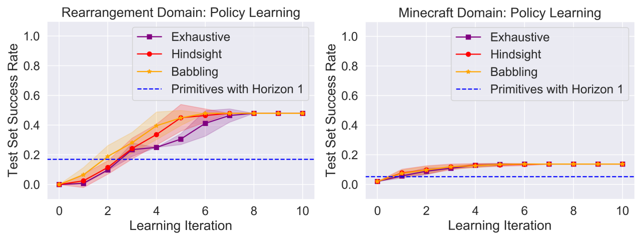


Figure 6: Test set performance versus learning iteration for invoking learned, lifted policies to solve single-fluent goals in the test set. Each learning iteration provides the agent with only a *single* trajectory of data. In both domains, the policies we learn quickly start to outperform invoking only the primitives (represented by the blue dashed horizontal line). To improve performance further, learn operators for these policies and plan with them; see Figure 7.

that are overly specific to the initial states it has seen so far, which due to the exhaustive search for novelty make up a very small portion of the space of initial states in $P(I)$. Therefore, it takes more iterations to train policies under the *Exhaustive* strategy; however, in environments where some important goals are very rare within $P(G)$, this data-collection strategy could outperform goal-babbling.

Limitations and Future Work

A major limitation of our current approach is that planning with the learned operators for policies is significantly slower than planning with only the learned operators for primitives. In a head-to-head comparison where both approaches are bound in terms of computation time, rather than planning horizon, the primitives alone fare better than our learned operators. For instance, on later iterations of learning in the *Minecraft* domain, planning with the learned policies takes around 3 seconds per problem, while planning with primitives takes around 0.1 seconds per problem. The reason for this slowdown is that each policy itself grows to be complex very quickly, as it must capture a variety of scenarios seen in the training data, leading to less efficient STRIPS planning. For example, a policy for `Holding` an

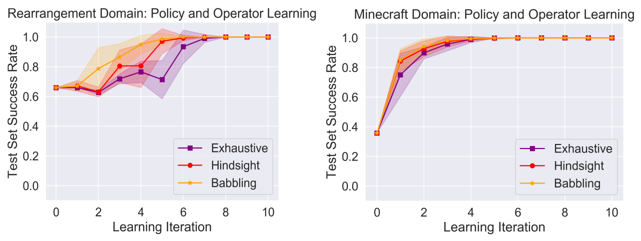


Figure 7: Test set performance versus learning iteration for planning with learned goal-conditioned policies and associated operators. The planner is resource-limited with a maximum horizon of 5. In both domains, it is clear that the best performance is achieved by this unified approach.

object would have to reason about whether it is obstructed.

To alleviate this issue, one option is to turn to parametric models, whose complexity would not grow (as logical decision trees do) with the amount of training data. Another option is to pursue aggressive *pruning* strategies at the level of training data, individual skills, or sets of skills (Minton 1985). Training data could be pruned by selecting only subtrajectories from demonstrations, e.g., those that are most common or most representative by some metric. Individual policies could be pruned by removing preconditions or effects from operators or by regularizing the policies themselves. Sets of policies could be pruned by removing policies that appear to hinder planning more than they help.

Scaling up our experiments to real-world domains will require innovating in several directions. To start, we intend to generalize the methods presented here to stochastic domains, which would likely require moving to probabilistic inductive logic programming systems (De Raedt and Kersting 2008) as our learning algorithm. This would also allow the system to be robust to both noise and inconsistent action selection in the training data.

To continue building toward a general-purpose hierarchy, two more major improvements are necessary to our architecture. First, learned policies should be able to invoke other learned policies. Second, the primitive operators should be allowed to improve while policies are being learned, since it is unreasonable in larger domains for random exploration to be enough to train completely correct primitive operators. In both of these cases, an important question arises: *how should higher levels of the hierarchy be adapted when a lower level operator is found to be incorrect*? A very simple, domain-independent strategy would be to discard higher levels of the hierarchy when a lower-level update occurs, and re-build these higher levels from new data; we hope to consider more sophisticated approaches in the future.

Furthermore, we hope to investigate other data-collection strategies, such as purposefully setting goals that are not too easy based on the agent's current skillset (Chaiklin 2003), and especially to allow training on data that fails to reach these goals, since useful signal can still be extracted here.

References

Alcázar, V.; Borrajo, D.; Fernández, S.; and Fuentetaja, R. 2013. Revisiting regression in planning. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

Baranes, A., and Oudeyer, P.-Y. 2010. Intrinsically motivated goal exploration for active motor learning in robots: A case study. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1766–1773. IEEE.

Blockeel, H., and De Raedt, L. 1998. Top-down induction of first-order logical decision trees. *Artificial intelligence* 101(1-2):285–297.

Chaiklin, S. 2003. The zone of proximal development in vygotsky's analysis of learning and instruction. *Vygotsky's educational theory in cultural context* 1:39–64.

Coumans, E.; Bai, Y.; and Hsu, J. 2018. Pybullet physics engine.

De Raedt, L., and Kersting, K. 2008. Probabilistic inductive logic programming. In *Probabilistic Inductive Logic Programming*. Springer. 1–27.

De Raedt, L. 1996. Induction in logic. *Proceedings of the 3rd International Workshop on Multistrategy Learning* 1:29–38.

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.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial intelligence* 3:251–288.

Florensa, C.; Held, D.; Wulfmeier, M.; Zhang, M.; and Abbeel, P. 2017. Reverse curriculum generation for reinforcement learning. *arXiv preprint arXiv:1707.05300*.

Groshev, E.; Tamar, A.; Goldstein, M.; Srivastava, S.; and Abbeel, P. 2018. Learning generalized reactive policies using deep neural networks. In *AAAI Spring Symposium*.

Held, D.; Geng, X.; Florensa, C.; and Abbeel, P. 2018. Automatic goal generation for reinforcement learning agents.

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

Hoffmann, J. 2001. Ff: The fast-forward planning system. *AI magazine* 22(3):57–57.

Jinnai, Y.; Abel, D.; Hershkowitz, D.; Littman, M.; and Konidaris, G. 2019. Finding options that minimize planning time. In *Proceedings of the 36th International Conference on Machine Learning*.

Johnson, M.; Hofmann, K.; Hutton, T.; and Bignell, D. 2016. The malmo platform for artificial intelligence experimentation. In *IJCAI*, 4246–4247.

Kaelbling, L. P., and Lozano-Pérez, T. 2010. Hierarchical planning in the now. In *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*.

Konidaris, G., and Barto, A. G. 2007. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, 895–900.

Konidaris, G., and Barto, A. G. 2009. Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in neural information processing systems*, 1015–1023.

Lavrac, N., and Dzeroski, S. 1994. Inductive logic programming. In *WLP*, 146–160. Springer.

Minton, S. 1985. Selectively generalizing plans for problem-solving. 596 – 599.

Mourao, K.; Zettlemoyer, L. S.; Petrick, R.; and Steedman, M. 2012. Learning strips operators from noisy and incomplete observations. *arXiv preprint arXiv:1210.4889*.

Muggleton, S. 1991. Inductive logic programming. *New generation computing* 8(4):295–318.

Oh, J.; Singh, S.; Lee, H.; and Kohli, P. 2017. Zero-shot task generalization with multi-task deep reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 2661–2670. JMLR. org.

Peters, J., and Schaal, S. 2008. Reinforcement learning of motor skills with policy gradients. *Neural networks* 21(4):682–697.

Pollock, J. L. 1998. The logical foundations of goal-regression planning in autonomous agents. *Artificial Intelligence* 106(2):267–334.

Quinlan, J. R. 1986. Induction of decision trees. *Machine learning* 1(1):81–106.

Quinlan, J. R. 1990. Learning logical definitions from relations. *Machine learning* 5(3):239–266.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature* 529(7587):484.

Stolle, M., and Precup, D. 2002. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, 212–223. Springer.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence* 112(1-2):181–211.

Tesauro, G. 1995. Temporal difference learning and td-gammon. *Communications of the ACM* 38(3):58–68.

Tessler, C.; Givony, S.; Zahavy, T.; Mankowitz, D. J.; and Mannor, S. 2017. A deep hierarchical approach to lifelong learning in minecraft. In *Thirty-First AAAI Conference on Artificial Intelligence*.

Thrun, S., and Schwartz, A. 1995. Finding structure in reinforcement learning. In *Advances in neural information processing systems*, 385–392.

Wang, Z.; Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2018. Active model learning and diverse action sampling for task and motion planning. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 4107–4114. IEEE.

Wang, X. 1996. Planning while learning operators. In *AIPS*, 229–236.