Learning Symbolic Operators for Task and Motion Planning

Tom Silver*, **Rohan Chitnis***, **Joshua Tenenbaum, Leslie Pack Kaelbling, Tomás Lozano-Pérez**
MIT Computer Science and Artificial Intelligence Laboratory
{tslvr, ronuchit, jbt, lpk, tlp}@mit.edu

***Abstract*— Robotic planning problems in hybrid state and action spaces can be solved by integrated task and motion planners (TAMP) that handle the complex interaction between motion-level decisions and task-level plan feasibility. TAMP approaches rely on domain-specific symbolic operators to guide the task-level search, making planning efficient. In this work, we formalize and study the problem of operator learning for TAMP. Central to this study is the view that operators define a lossy abstraction of the transition model of a domain. We then propose a bottom-up relational learning method for operator learning and show how the learned operators can be used for planning in a TAMP system. Experimentally, we provide results in three domains, including long-horizon robotic planning tasks. We find our approach to substantially outperform several baselines, including three graph neural network-based model-free approaches from the recent literature. Video: `https://youtu.be/iVfpX9BpBRo`. Code: `https://git.io/JCT0g`**

I. Introduction

Robotic planning problems are often formalized as *hybrid* optimization problems, requiring the agent to reason about both discrete and continuous choices (e.g., *Which object should I grasp?* and *How should I grasp it?*) [1]. A central difficulty is the complex interaction between low-level geometric choices and high-level plan feasibility. For example, how an object is grasped affects whether or not it can later be placed into a shelf (Figure 1). Task and motion planning (TAMP) combines insights from AI planning and motion planning to address these challenges [1], [2], [3], [4], [5], [6]. TAMP uses symbolic planning operators to search over symbolic plans, biasing the search over motions. Operators are hand-specified in all popular TAMP systems [7], requiring expert input for each domain. Instead, we aim to develop a domain-independent operator learning algorithm for TAMP.

Symbolic operators are useful for TAMP in three key ways. First, operators let us efficiently determine that many plans have zero probability of reaching a goal, regardless of the choice of continuous action parameters, allowing us to ignore such plans in the search. Second, operators permit a bilevel optimization approach, with symbolic planning providing a dense sequence of subgoals for continuous optimization [1], [5]. Third, explicit PDDL-style operators [8] allow us to automatically derive AI planning heuristics, which dramatically speed up the search over symbolic plans.

In this work, we formalize and study the problem of learning operators for TAMP from data, using a bottom-up relational learning method. We consider a setting with a deterministic low-level environment simulator, a set of hybrid controllers with associated samplers for the continuous parameters, an object-oriented continuous state, and a set of predicates that collectively define a lossy abstraction of the low-level state. In this setting, planning is possible *without* operators via breadth-first search over sequences of controllers and a schedule for calling the associated samplers (Section VI, Baseline 5). However, we find that making use of the operators within TAMP provides enormous benefits.

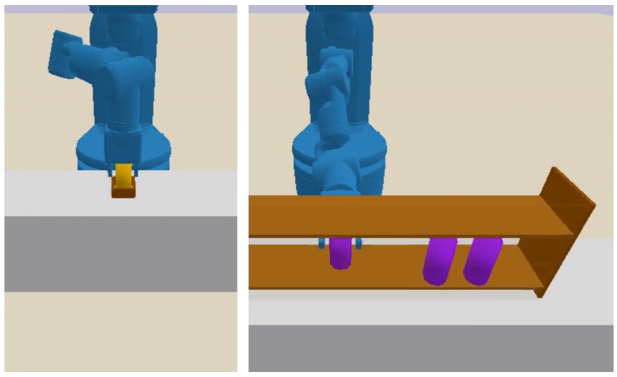

Fig. 1: Snapshots from the "Painting" domain (Section VI). *Left:* An object that is grasped from the top can be placed into a small open-faced box. *Right:* A side grasp is required to place an object into a shelf with a ceiling. The chosen continuous grasp argument for the `Pick` action therefore later influences the effects of placing.

Our approach continues a line of recent work that seeks to exploit properties of task distributions to make TAMP more efficient. In particular, our approach can be understood as a model-based method for learning guidance in hybrid planning problems: the operators define an abstract transition model that provides guidance. Recent literature on learning for TAMP considers various model-free counterparts. Kim and Shimanuki [9] learn an "abstract Q-function" that implicitly defines a goal-conditioned policy over symbolic actions, while Driess et al. [10] learn a recurrent model that directly produces symbolic plans. In our experiments, we consider three model-free baselines inspired by these methods.

In this paper, we propose the Learning Operators For TAMP (LOFT) algorithm, and make the following contributions: (1) we formalize the problem of operator learning in TAMP; (2) we propose a relational operator learning method, and show how to use the learned operators to quickly solve TAMP problems; (3) we provide experimental results in three domains, including long-horizon robotic planning tasks, that show the strength of our approach over several baselines.

* Equal contribution.

II. Related Work

A. Task and Motion Planning (TAMP)

The field of TAMP emerged from the combination of AI methods for task planning [11] and robotic methods for motion planning. TAMP methods are focused on solving multimodal, continuous robotic planning problems in highly unstructured environments [12], [13]; see Garrett et al. [1] for a recent survey. Approaches to TAMP can be broadly categorized based on how the treatment of symbolic reasoning interacts with the treatment of continuous variables during planning. Some approaches to TAMP involve optimization over trajectories [3], [6], while others use sampling-based procedures [4], [5], [14]. In all cases, popular TAMP systems rely on hand-specified planning models (e.g., PDDL operators), a limitation we aim to address in this paper.

B. Learning for Task and Motion Planning

Learning techniques have been integrated into many aspects of TAMP systems, from learning samplers for continuous values [15], [16], [17], [18] to learning guidance for symbolic planning [9], [10], [16]. The latter is our focus in this paper; we assume samplers are given, and we aim to learn operators that enable symbolic planning in TAMP.

Most relevant to our work are efforts to learn model-free search guidance for symbolic planning [9], [10]. A challenge in applying model-free techniques in the TAMP setting is that there is no obvious way to "execute" an action in the space of symbolic transitions. Kim and Shimanuki [9] address this challenge by sampling low-level transitions at each step during symbolic planning; Driess et al. [10] instead learn a recurrent "Q-function" that takes in a sequence of actions and an *initial* state. In our experiments, we consider three baselines inspired by these model-free approaches.

C. Learning Symbolic Operators

Learning symbolic planning operators has a very long history in the planning literature; see Arora et al. [19] for a comprehensive review. While it has been studied for decades [20], [21], [22], [23], [24], operator learning has not been studied in the TAMP setting, where the learned operators must be understood as a lossy and abstract description of a low-level, geometric planning problem. In TAMP, the operators are a means to an end, not the entire story: the operators enable symbolic planning, which in turn produces candidate symbolic plans for a low-level optimizer. The operators in TAMP, therefore, are useful in that they give guidance to the overall planning procedure.

III. Problem Setting

We consider a standard TAMP setting with low-level states $x \in \mathcal{X}$, where x is a mapping from a set of typed objects O to attributes and their corresponding values. The attributes are fixed for each object type, and all values are real-valued vectors. For example, x may include the continuous 6D pose of each object in the scene. We are given a set of hybrid controllers $\Pi = \{\pi_1, \ldots, \pi_k\}$, each parameterized by zero or more discrete objects $\overline{o} = (o_1, \ldots, o_m)$ with $o_i \in O$, and by a real-valued continuous vector parameter $\overline{\theta}$. An action a is an instantiation of a controller with arguments, both discrete and continuous. For example, the action $\texttt{Pick(obj1,}\theta\texttt{)}$ is a call to a controller $\texttt{Pick} \in \Pi$ that will attempt to pick the object $o_1 \leftarrow \texttt{obj1}$ using grasp arguments $\overline{\theta} \leftarrow \theta$.

Each controller is associated with a given *sampler* for the continuous parameters $\overline{\theta}$, conditioned on the low-level state and discrete arguments. Samplers produce values that satisfy implicit constraints specific to their controller; for example, the sampler for the `Pick` controller produces feasible grasps θ based on the state x and the discrete argument o_1.[1]

We are given a low-level simulator f defining the environment dynamics. f maps a low-level state and an action to a next low-level state, denoted $x_{t+1} = f(x_t, a_t)$. We do not assume any analytical knowledge of the transition model.

We assume access to a set of predicates P. Each predicate $p(\overline{o}) \in P$ represents a named relation among one or more objects in O. For example, $\texttt{On(}y, z\texttt{)}$ encodes whether an object y is on top of another object z. In this work, all predicates are discrete: arguments are objects, and predicates either hold or do not hold (*cf.* numeric fluents). A predicate with variables as arguments is *lifted*; a predicate with objects as arguments is *ground*. Each predicate is a classifier over the low-level state x. Given a state x, we can compute the set of all ground predicates that hold in the state, denoted $s = \textsc{Parse}(x)$, where Parse is a deterministic function. For example, Parse may use the geometric information in the low-level state x to determine which objects are on other objects, adding $\texttt{On(}y, z\texttt{)}$ to s if object y is on object z. We refer to s as the *symbolic state*. The assumption that predicates are provided is limiting but standard in the learning-for-TAMP literature [9], [10], [15]. We emphasize that the predicates are *lossy*, in the sense that transitions at the symbolic level can be non-deterministic, even though the low-level simulator f and Parse function are deterministic [25].

We are given a set of planning problems $\{(O, x_0, G)\}$ to solve. Here, O is an object set, $x_0 \in \mathcal{X}$ is an initial low-level state, and G is a goal. All problems share a simulator f, predicates P, and controllers Π. A goal G is a (conjunctive) set of ground predicates over the object set O; we say G is *achieved* in state x if $G \subseteq \textsc{Parse}(x)$. A solution to a planning problem is a *plan*: a sequence of actions $(a_0, \ldots, a_{T-1})$ where $x_{t+1} = f(x_t, a_t)$ and x_T achieves G.

Since we are interested in learning-based approaches, we suppose that we are given a dataset $\mathcal{D}$, collected offline, of low-level transitions (x_i, a_i, x_{i+1}, G_i) generated from planning in (typically smaller) problems from the same family. See Section VI for details on how we collect $\mathcal{D}$ in practice.

IV. Task and Motion Planning with Operators

Most TAMP systems rely on hand-defined, domain-specific symbolic *operators* to guide planning. In this section, we define operators and describe how they are used for planning in "search-then-sample" TAMP methods [1].

[1]Not all TAMP systems use samplers; some are optimization-based [6]. Our approach is not limited to sampling-based TAMP systems, but we find it convenient for exposition to describe our problem setting this way.

A symbolic operator is composed of a controller, parameters, a precondition set, and an effect set.[2] The parameters are typed placeholders for objects that are involved in the discrete controller parameters, the precondition set, or the effect set. Preconditions are lifted predicates over the parameters that describe what must hold for the operator to be applicable. Effects are (possibly negated) lifted predicates over the parameters that describe how the symbolic state changes as a result of applying this operator (executing this controller). The operator can be *grounded* by assigning the parameters to objects, making substitutions in the parameters, preconditions, and effects accordingly. In this paper, operators do not model the influence of the continuous action parameters, and therefore make predictions based on the discrete controller parameters and symbolic state alone. See the third panel of Figure 2 for an example operator (`Pick0`).

To understand how operators can be used to guide TAMP, we turn to the following definitions.

Definition 1 (Action template): An *action template* is a controller $\pi(\overline{o}, \overline{\theta})$ and an assignment of the controller's discrete parameters $\overline{o} \leftarrow o$, with the continuous parameters left unassigned. We denote the action template as $\pi(o, \cdot)$.

An action template can be understood as an action with a "hole" for the continuous arguments of the controller. For example, `Pick(obj1,·)` is an action template with a hole left for the continuous grasp arguments.

Definition 2 (Plan skeleton): A *plan skeleton* is a sequence of action templates $(\pi_1(o_1, \cdot), \ldots, \pi_\ell(o_\ell, \cdot))$.

A plan skeleton can be *refined* into a plan by assigning values to all of the continuous parameters in the controller: $(\pi_1(o_1, \theta_1), \ldots, \pi_\ell(o_\ell, \theta_\ell)) = (a_1, \ldots, a_\ell)$. The main role of operators is to efficiently generate plan skeletons that can be refined into a solution plan. Given a goal G and an initial low-level state x_0 with symbolic state $s_0 = \textsc{Parse}(x_0)$, we can use the operators to search for a plan skeleton that achieves G symbolically, before needing to consider any continuous action arguments. Importantly, though, a plan skeleton that achieves G symbolically has no guarantee of being refinable into a plan that achieves G in the environment. This complication is due to the lossiness of the symbolic abstraction induced by the predicates.

To address this complication, search-then-sample TAMP methods perform a bilevel search, alternating between *high-level symbolic planning* to search for plan skeletons and *low-level optimization* to search for continuous arguments that turn a plan skeleton into a valid plan [2], [3], [5].

In this work, for high-level planning, we use A* search. Importantly, our operators are compatible with PDDL representations [8], meaning we can use classical, domain-independent planning heuristics in this search (hAdd [11] in all experiments). For low-level optimization, we conduct a backtracking search over continuous parameter assignments to attempt to turn a plan skeleton into a plan. Recall that each step in the plan skeleton is an action template, and that each controller is associated with a sampler. For each action template, we invoke the sampler to produce continuous arguments, which in turn leads to a complete action. The backtracking is conducted over calls to the sampler for each step of the plan skeleton. This search terminates with success if following the action sequence in the simulator f results in a low-level state that achieves the goal. We allow a maximum of N_{samples} calls to each step's sampler before backtracking. If search is exhausted, control is returned to the high-level A^* search to produce a new candidate plan skeleton.

A major benefit of having symbolic operators is that they allow us to prune the backtracking search by only considering plan prefixes that induce trajectories in agreement with the expected symbolic trajectory. For example, consider a plan prefix (a_0, a_1, a_2), which induces the trajectory (x_0, x_1, x_2, x_3). We can call PARSE on each state to produce the symbolic state sequence (s_0, s_1, s_2, s_3); if the transition $s_2 \rightarrow s_3$ is not possible under the operators, then the continuous arguments of a_2 should be resampled.

The TAMP algorithm described above was selected due to its relative simplicity and strong empirical performance [5]. Note, however, that it is *not* probabilistically complete; it is possible that the low-level optimization cannot refine a valid skeleton within the N_{samples} allotment, and since there is no ability to revisit a skeleton once low-level optimization fails, the algorithm may miss a solution even if one exists.

V. Learning Symbolic Operators for TAMP

In this section, we describe our main approach: Learning Operators For TAMP, or LOFT for short. We begin in Section V-A with the observation that we can learn *probabilistic* operators to account for the inevitable lossiness induced by the symbolic abstraction. In Section V-B, we describe an efficient algorithm for learning these probabilistic operators from transition data. Finally, in Section V-C we demonstrate how to use the learned probabilistic operators for TAMP. See Figure 2 for an example of the full pipeline.

A. Probabilistic Operators as Symbolic Transition Models

From a planning perspective, operators can be seen as a substrate for guiding search. An alternative perspective that is more amenable to learning is that the operators comprise a *symbolic transition model*, describing the distribution of next symbolic states s_{t+1} given a current symbolic state s_t and action a_t: $P(s_{t+1} \mid s_t, a_t)$.[3] This probabilistic framing follows from the fact that no deterministic function could accurately model these symbolic transitions, given the lossiness of the abstraction induced by PARSE.

To learn operators that comprise a probabilistic transition model, we consider operators with *probabilistic effects*, like those found in PPDDL [26]. These probabilistic operators

[2]Each controller can be associated with *multiple* operators. For example, in the Painting domain shown in Figure 1, a generic `Place` controller would have two operators, one for placing into the shelf (with `HoldingSide` in the preconditions and `InShelf` in the effects) and another for placing into the box (with `HoldingTop` and `InBox` respectively).

[3]In general, the nondeterminism in the symbolic transitions may depend on the policy for generating the transitions and the low-level dynamics. For our purposes, the precise semantics are unimportant as the probabilities are only used to filter out rare outcomes, and are then discarded (Section V-C).

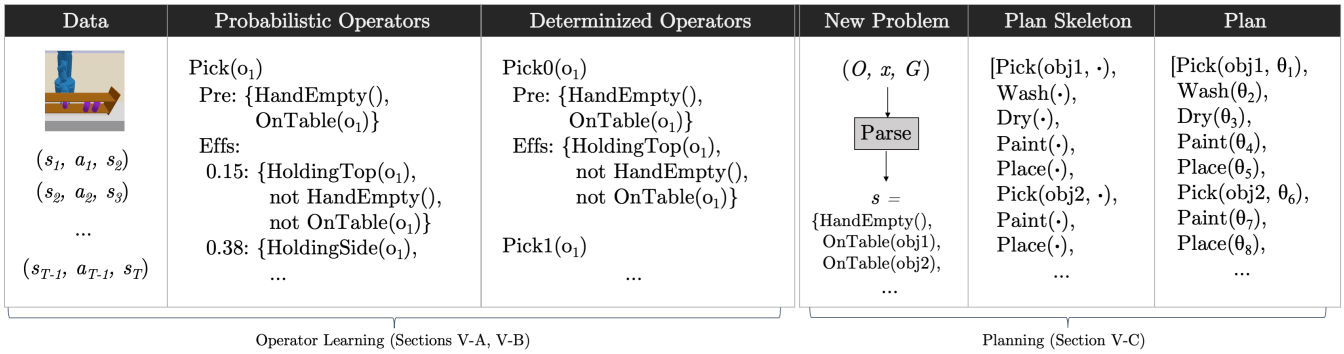

Fig. 2: An example of the complete pipeline. We learn probabilistic operators on transition data, then determinize them. The operators are used to generate high-level plan skeletons for new TAMP problem instances, providing guidance that makes planning more efficient.

Algorithm PROBABILISTIC OPERATOR LEARNING

```
// Transition data for a single controller
Input: D_π = {(s_i, a_i, s_{i+1}) : a_i = π(·,·)}
// Cluster data by lifted effect sets
clusters ← CLUSTERLIFTEDEFFECTS(D_π)
// Compute and save precondition sets
effectsToPreconditionSets ← {}
for D_(π,eff) ∈ clusters do
    // Learn one or more precondition sets
    effectsToPreconditionSets[eff] ←
      LEARNPRECONDITIONSETS(D_(π,eff), D_π)
// Instantiate operators
operators ← ∅
for each seen precondition set pre do
    // Effects with these preconditions
    effs ← {eff : pre ∈
      effectsToPreconditionSets[eff]}
    operator ← MAKEOPERATOR(π, pre, effs)
    operators.add(operator)
// Estimate effect probabilities
for operator ∈ operators do
    ESTIMATEPARAMETERS(operator, D_π)
return operators
```

Algorithm 1: Algorithm for learning probabilistic operators for a given controller π. See Section V-B for details.

are equivalent to the deterministic operators described in Section IV, except that they contain a categorical distribution over effect sets rather than a single effect set. See the second panel in Figure 2 for an example (`Pick`).

B. Learning Probabilistic Operators

We now describe a bottom-up relational method for learning probabilistic operators in the TAMP setting. Recall that we are given a dataset $\mathcal{D} = \{(x_i, a_i, x_{i+1}, G_i)\}$ of low-level transitions. (The goals are irrelevant for learning a transition model, but they are included in $\mathcal{D}$ because they prove useful for baselines in our experiments.) These data can be converted into symbolic transitions $\{(s_i, a_i, s_{i+1})\}$ by calling the PARSE function on the low-level states. We now have samples from the distribution $P(s_{t+1} \mid s_t, a_t)$ that we wish to learn. The data can be further partitioned by controller. Let $\mathcal{D}_\pi$ denote the dataset of transitions for controller $\pi \in \Pi$, e.g., $\mathcal{D}_{\texttt{Pick}} = \{(s_i, a_i, s_{i+1}) \in \mathcal{D} : a_i = \texttt{Pick}(\cdot,\cdot)\}$. With these datasets in hand, the algorithm for learning probabilistic operators proceeds in three steps: (1) lifted effect clustering, (2) precondition learning, and (3) parameter estimation. See Algorithm 1 for pseudocode.

Lifted Effect Clustering. We begin by clustering the transitions in $\mathcal{D}_\pi$ according to lifted effects. For each transition (s_i, a_i, s_{i+1}), we compute ground effects using two set differences: $s_{i+1} - s_i$ are positive effects, and $s_i - s_{i+1}$ are negative effects. We then cluster pairs of transitions together if their effects can be *unified*, that is, if there exists a bijective mapping between the objects in the two transitions such that the effects are equivalent up to this mapping. This unification can be checked in time linear in the sizes of the effect sets. Each of the resulting clusters is labelled with the lifted effect set, where the objects in the effects from any arbitrary one of the constituent transitions are replaced with placeholders. Algorithm 1 uses the notation $\mathcal{D}_{(\pi,\text{eff})}$ to denote the dataset for controller π and lifted effect set "eff."

Precondition Learning. Next, we learn one or more sets of preconditions for each lifted effect cluster. We perform two levels of search: an outer greedy search over sets of preconditions, and an inner best-first search over predicates to include in each set. The outer search is initialized to an empty set and calls the inner search to generate successors one at a time, accepting all successors until terminating after a maximum number of steps or failure of the inner search.

The inner search for a single precondition set is initialized by lifting each previous state s_i for all transitions (s_i, a_i, s_{i+1}) in the effect cluster; each object in s_i is replaced with a placeholder variable, and the resulting predicate set represents a (likely over-specialized) candidate precondition. Successors in this inner search are generated by removing each possible precondition from the current candidate set.

A transition is considered *explained* by a precondition set if there exists some substitution of the precondition variables to the objects in the transition so that the effects are equivalent under the substitution to that of the current effect cluster. A precondition set is desirable if it leads to many "true positive" transitions — ones that are explained

by this precondition set, but not by any previously selected ones. A precondition set is undesirable if it leads to many "false positive" transitions, where the preconditions hold under some variable substitution, but the effects do not match the cluster. We therefore assign each candidate set a weighted sum score (higher is better): $\beta \times$ (# true positives) $-$ (# false positives), where β is a hyperparameter ($\beta = 10$ in all experiments). The inner search terminates after a maximum number of iterations or when no improving successor can be found. The highest-scoring candidate is returned.

Precondition learning is the computationally hard step of the overall algorithm; the outer and inner searches are approximate methods for identifying the best sets of preconditions under the score function. The computational complexity is bounded by the number of iterations in the inner search (100 in all experiments), the amount of data, the number of controllers, and the number of predicates in the largest state.

Parameter Estimation. After precondition learning, we have one or more sets of preconditions for each set of lifted effects, for each controller. Looking between lifted effects, it will often be the case that the same precondition set (up to unification) appears multiple times. In the example of Figure 2 for `Pick`, the precondition set $\{$`HandEmpty()`, `OnTable(`o_1`)`$\}$ may be associated with two sets of effects, one that includes `HoldingSide(`o_1`)` and another that includes `HoldingTop(`o_1`)`. We combine effects with matching preconditions to initialize the probabilistic operators.

All that remains is to estimate the probabilities associated with each operator's effect sets. This parameter estimation problem reduces to standard categorical distribution learning; for each pair of lifted preconditions and effects, we count the number of transitions for which the lifted preconditions hold and the number for which the effects follow, and divide the latter by the former. This results in a final set of probabilistic operators, as shown in the second panel of Figure 2.[4]

In terms of taxonomy, our proposed algorithm can be seen as a bottom-up inductive logic programming (ILP) algorithm, although it falls outside of the typical problem setting considered in ILP [27]. In the operator learning literature [19], a close point of comparison is the algorithm of Zettlemoyer, Pasula, and Kaelbling [20], who learn "noisy deictic rules" that can be converted into probabilistic operators. We compare our method against learning noisy deictic rules (LNDR) in experiments, finding ours to be faster and more effective for our domains. This difference stems from our decoupling of effect clustering, precondition learning, and parameter estimation, which are all interleaved in LNDR.

C. Planning with Learned Operators

We now describe how the learned probabilistic operators can be used for TAMP. In Section IV, we described how *deterministic* operators can be used for solving TAMP problems. A simple and effective approach for converting probabilistic operators into deterministic ones is *all-outcome determinization* [28], whereby one deterministic operator is created for each effect set in each probabilistic operator, with the corresponding preconditions and parameters (Figure 2, third panel). Before determinizing, we filter out effects that are highly unlikely by thresholding on a hyperparameter p_{min}.

In the limiting case where the dataset $\mathcal{D}$ is empty and we were unable to learn any operators, planning reduces to simply invoking the backtracking search on every possible plan skeleton, starting from length-1 sequences, then length-2, etc. We use this strategy as a baseline (B5) in our experiments. Since planning is still possible even without operators, it is clear that the learned operators should be understood as providing *guidance* for solving TAMP problems efficiently; with more data, the guidance improves.

Our aim is not to innovate on TAMP, but rather to demonstrate that the operators underpinning TAMP methods can be learned from data. In addition to our main TAMP algorithm, we also found preliminary success in using LOFT with another popular TAMP system [14], but stayed with the method described in Section IV due to speed advantages.

VI. Experiments

A. Experimental Setup

Domains. We conduct experiments in three domains.

"Cover" is a relatively simple domain in which colored blocks and targets with varying width reside along a 1-dimensional line. The agent controls a gripper that can pick and place blocks along this line, and the goal is to completely cover each target with the block of the same color. The agent can only pick and place within fixed "allowed regions" along the line. Because the targets and blocks have certain widths, and because of the allowed regions constraint, the agent must reason in advance about the future placement in order to decide how to grasp a block. There are two object types (`block` and `target`), and the predicates are `Covers(?block, ?target)`, `Holding(?block)`, and `HandEmpty()`. There are two controllers, `Pick(?block, ?loc)` and `Place(?targ, ?loc)`, where the second parameter of each is a continuous location along the line. The samplers for both actions choose a point uniformly from the allowed regions. We evaluate the agent on 30 randomly generated problems per seed, with average optimal plan length 3.

"Blocks" is a continuous adaptation of the classical blocks world planning problem. A robot uses its gripper to interact with blocks on a tabletop and must assemble them into various towers. See Figure 3 for a snapshot. All geometric reasoning, such as inverse kinematics and collision checking, is implemented through PyBullet [29]. The predicates are `On(?b1, ?b2)`, `OnTable(?b)`, `Clear(?b)`, `Holding(?b)`, and `HandEmpty()`. There are three controllers, `Pick(?b)`, `Stack(?b)`, and `PutOnTable(?loc)`, where `PutOnTable` is parameterized by a continuous value describing where on the table to place the currently held block. This place location must

[4]These learned operators may not represent a proper probability distribution, since there is a possibility that one transition will be fit by multiple preconditions; see [20] for further discussion. For our purposes, the probabilities serve only to filter out low-probability effects on the way to guiding planning (Section V-C), so this technicality can be ignored.

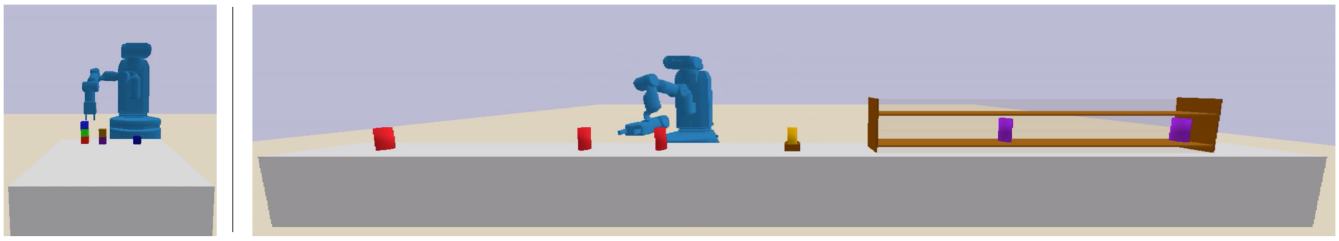

Fig. 3: Snapshots from the "Blocks" domain (left) and the "Painting" domain (right). See Section VI for details.

be chosen judiciously, because otherwise it could adversely affect the feasibility of the remainder of a plan skeleton. The sampler for the `PutOnTable` controller is naive, randomly sampling a reachable location on the table while ignoring obstacles. We evaluate the agent on 10 randomly generated problems per seed, with average optimal plan length 8.

"Painting" is a challenging, long-horizon robotic planning problem, in which a robot must place objects at target positions located in either a shelf or a box. Before being placed, objects must first be washed, dried, and painted with a certain color. Objects can be randomly initialized to start off clean or dry. To place into the shelf, the robot must first side-grasp the object due to the shelf's ceiling; similarly with top-grasping for the box. This introduces a dependency between the grasp parameter and the feasibility of placing up to four timesteps later. See Figure 3 for a snapshot, where the yellow object is in the box and the purple objects are in the shelf. All geometric reasoning is implemented through PyBullet [29]. There are 14 predicates: `OnTable`, `Holding`, `HoldingSide`, `HoldingTop`, `InShelf`, `InBox`, `IsDirty`, `IsClean`, `IsDry`, `IsWet`, `IsBlank`, `IsShelfColor`, `IsBoxColor`, all parameterized by a single `?obj`, and `HandEmpty()`. There are five controllers, `Pick(?obj, ?base, ?grip)`, `Place(?base, ?grip)`, `Wash(?obj, ?effort)`, `Dry(?obj, ?effort)`, and `Paint(?color)`. Here, `?effort` and `?color` are continuous values in $\mathbb{R}$, and `?base` and `?grip` are continuous values in $\mathbb{R}^3$ denoting base and end effector positions that the controller should attempt to go to before executing the pick or place. The `Pick` sampler randomly returns a top grasp or a side grasp. The `Place` sampler is bimodal: with probability 0.5, it samples a random placement in the shelf; otherwise, it samples a random placement in the box. As in Blocks, the placement samplers are naive with respect to potential collisions. The samplers for `Wash` and `Dry` are degenerate, returning exactly the appropriate `?effort` required to wash or dry the object respectively. The `Paint` sampler randomly returns the shelf color or the box color. We evaluate the agent on 30 randomly generated problems per seed, with average optimal plan length 31.

Methods Evaluated. We evaluate the following methods.

- LOFT: Our full approach.
- Baseline B0: We use LOFT but replace our Algorithm 1 for probabilistic operator learning with LNDR [20], a popular greedy algorithm for learning noisy rules.
- Baseline B1: Inspired by Kim and Shimanuki [9], we train a graph neural network (GNN) that represents a Q-function for high-level search. The model takes as input a symbolic state, an action template, and the goal, and outputs a Q-value for that action template. To use the model, we perform tree search as described in [9]: the Q-function is used to select the best action template, and we sample W (a width parameter) continuous values for calling the simulator to produce successor states. A major difference from their work is that we are not doing online learning, so we perform fitted Q-iteration with the given dataset $\mathcal{D}$ in order to train the GNN.
- Baseline B2: Same as B1, but the low-level state x is also included in the input to the GNN.
- Baseline B3: Inspired by Driess et al. [10], we train a recurrent GNN that predicts an entire plan skeleton conditioned only on the initial low-level state, initial symbolic state, and goal. This recurrent model sequentially predicts the next action template to append onto the skeleton. For each skeleton, we use our backtracking search method (Section IV) to attempt to optimize it.
- Baseline B4: We train a raw GNN policy that maps low-level states x to actions a. The GNN outputs a discrete choice of which controller to use, and values for both the discrete and continuous arguments of that controller. This baseline does not make use of the samplers.
- Baseline B5: As discussed in Section V-C, we run our planning algorithm with no operators, which reduces to trying to optimize every possible plan skeleton. This is the limiting case of LOFT where the dataset $\mathcal{D} = \emptyset$.
- Oracle: We use our planner with good, hand-written operators for each domain. This method represents an upper bound on the performance we can get from LOFT.

Data Collection. In all domains, we use the same data collection strategy. First, we generate a set of 20 problems from the domain that are smaller (with respect to the number of objects) than the ones used for evaluation. Then, we use an oracle planner to produce demonstrations of good behavior in these smaller problems. Finally, we collect "negative" data (which is important for learning preconditions) by, K times, sampling a random state x seen in the demonstrations, taking a random action a from that state, and seeing the resulting $x' = f(x, a)$, where f is our low-level simulator. We use $K = 100$ for Cover and Blocks, and $K = 2500$ for Painting.

The number of transitions $|\mathcal{D}|$ is 126 for Cover, 152 for Blocks, and 2819 for Painting. All learning-based methods

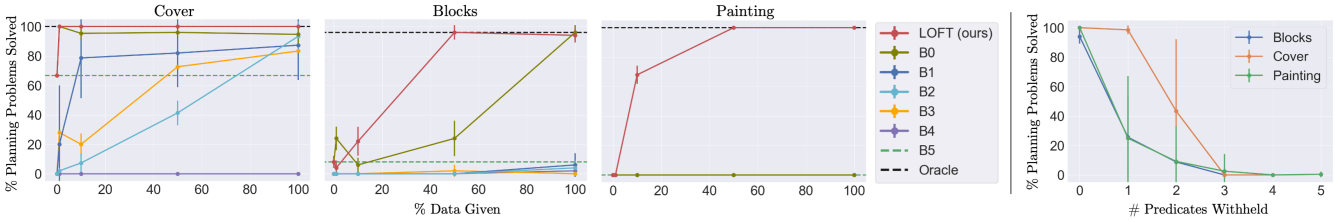

Fig. 4: *Left three:* For each of our domains, the percent of planning problems solved within a timeout, as a function of the amount of data provided to the approach. Each point is a mean over 5 seeds. Dotted lines indicate non-learning approaches (B5 and Oracle). Planning timeout was set to 1 second for Cover and 10 seconds for Blocks and Painting. In all domains, LOFT performs extremely well, reaching better planning performance than other approaches with less data. *Right:* Predicate ablation experiment, showing problems solved by LOFT as a function of the number of predicates withheld. Each point is a mean over 25 seeds; for each seed, we randomly select predicates to remove. LOFT is somewhat robust to a few missing predicates, but generally relies on being given a complete set to perform very well.

(LOFT and Baselines B0-B4) receive the exact same dataset.

Experimental Details. We use $p_{\text{min}} = 0.001$ and $N_{\text{samples}} = 10$ for all domains. We use a planning timeout of 1 second for Cover and 10 seconds for Blocks and Painting. For all GNN models, we preprocess literals to be arity 1 or 2 by either adding a dummy argument (if arity 0) or splitting into multiple literals (if arity > 2). All GNNs are standard encode-process-decode architectures [30], where node and edge modules are fully connected neural networks with one hidden layer of dimension 16, ReLU activations, and layer normalization. Message passing is performed for $K = 3$ iterations. Training uses the Adam optimizer with learning rate 0.001 and batch size 16. For B1 and B2, we use search width $W = 1$ for Cover and Blocks, and $W = 3$ for Painting; we use 5 iterations of fitted Q-iteration for Cover, and 15 for Blocks and Painting; and we train for 250 epochs per iteration. For B3 and B4, we train for 1000 epochs.

B. Results and Discussion

See Figure 4 (left three plots) for our main set of results, which show planning performance of each method as a function of the amount of data given. To assess performance with 50% of data given, for example, we train from scratch using only 50% of transitions randomly sampled from the full dataset $\mathcal{D}$. In all three domains, LOFT achieves the performance of the oracle with enough data. In Painting, none of the baselines are able to solve even a single planning problem within 10 seconds. Furthermore, across all domains, LOFT is more data-efficient than all learning-based baselines (especially the GNN ones), since it does not require as much data on average to learn probabilistic operators as it does to train a neural network.

The difference between LOFT and B0 (LOFT but with LNDR) is noteworthy: B0 often performs much worse. Inspecting the operators learned by LNDR, we find that the learning method is more liable to get stuck in local minima. For example, in the Painting domain, LNDR consistently uses one operator with the `Holding` predicate instead of two operators, one with `HoldingSide` and another with `HoldingTop`. Our learning method is able to avoid these pitfalls primarily because the lifted effect clustering is decoupled from precondition learning and parameter estimation. Our method learns effect sets once and does not revisit them until the operators are determinized during planning; in contrast, LNDR constantly re-evaluates whether to keep or discard an effect set in the course of learning operators.

Comparing B1 and B2 shows that model-free approaches can actually suffer from inclusion of the low-level state as input to the network, since this inclusion may make the learning problem more challenging. LOFT, on the other hand, does not use the low-level states to learn operators; this may be a limitation in situations where the predicates are not adequate for learning useful operators. Nevertheless, one could imagine combining model-free approaches like B1, B2, and B3 with LOFT: a learned Q-function could serve as a heuristic in our high-level A* search.

B4, the raw GNN policy, performs especially poorly in all domains since it does not make use of the samplers. This suggests that direct policy learning without hand-written samplers is challenging in many TAMP domains of interest.

LOFT outperforms B5 (planning without operators) given enough data. However, in Blocks, we see that with only a little data, LOFT actually performs slightly *worse* than B5, likely because with little data, the operators that are learned are quite poor and provide misleading guidance.

Table I reports training times for all methods. We see that LOFT trains extremely quickly — many orders of magnitude faster than the GNN baselines, which generally do not perform nearly as well as LOFT in our domains. This further confirms that learning symbolic operators is a useful and practical way of generating guidance for TAMP planners.

Finally, in two additional experiments, we measure the impact of ablating predicates on the performance of LOFT (Figure 4, rightmost plot) and test the importance of using classical heuristics in the high-level A* search (Table II). We can conclude from Figure 4 that LOFT has some robustness to missing predicates: even with up to 3 predicates withheld, LOFT remains the only non-oracle approach that solves any planning problems in Painting. However, performance deteriorates quickly; this is because we are using the operators in A* search with the hAdd heuristic, which is highly sensitive to missing preconditions or effects. Table II is an ablation study that shows the importance of using hAdd: our results are significantly worse if we instead use a blind heuristic, that is, a heuristic which is 0 everywhere. This speaks to the benefits of the fact that LOFT learns PDDL-style operators.

Training Times (seconds)						
Domain	LOFT **(ours)**	**B0**	**B1**	**B2**	**B3**	**B4**
Cover	0.13 (0.07)	0.47 (0.03)	131 (17)	145 (21)	497 (520)	127 (174)
Blocks	0.12 (0.07)	6.6 (0.42)	490 (48)	554 (72)	371 (122)	106 (146)
Painting	16 (0.91)	328 (21)	23704 (1033)	24326 (490)	5371 (3904)	816 (361)

TABLE I: Each entry shows the mean (standard deviation) training time in seconds over 5 seeds. Both LOFT and B0 train orders of magnitude faster than B1-B4. LOFT is also one order of magnitude faster than B0. B5 is excluded because it is not learning-based.

Heuristic Ablation Results		
Domain	LOFT **w/ hAdd**	LOFT **w/ blind**
Cover	100 (0)	100 (0)
Blocks	94 (5)	42 (12)
Painting	100 (0)	0 (0)

TABLE II: An ablation of the hAdd heuristic used in our method, where we replace it with a blind heuristic (always 0) that does not leverage the PDDL structure of the operators. Each entry shows the mean (standard deviation) percent of planning problems solved over 5 seeds. These results suggest that classical planning heuristics like hAdd are especially critical in our harder domains.

VII. Conclusion

We addressed the problem of learning operators for TAMP in hybrid robotic planning problems. The operators guide high-level symbolic search, making planning efficient. Experiments on long-horizon planning problems demonstrated the strength of our method compared to several baselines, including graph neural network-based model-free approaches.

A key future research direction is to relax the assumption that predicates are given. While assuming given predicates is standard in learning for TAMP, this is a severe limitation, since the quality of these predicates determines the quality of the abstraction. It would be useful to study what characterizes a good predicate, toward designing a predicate learning algorithm. Another useful future direction would be to learn the hybrid controllers that we are currently given, perhaps by pretraining with a reinforcement learning algorithm.

References

[1] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Perez, "Integrated task and motion planning," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, May 2021.

[2] F. Gravot, S. Cambon, and R. Alami, "aSyMov: a planner that deals with intricate symbolic and geometric problems," in *Robotics Research. The Eleventh International Symposium*. Springer, 2005.

[3] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, "Incremental task and motion planning: A constraint-based approach," in *Robotics: Science and Systems*, 2016, pp. 1–6.

[4] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2011, pp. 1470–1477.

[5] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 639–646.

[6] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

[7] Y. Zhu, J. Tremblay, S. Birchfield, and Y. Zhu, "Hierarchical planning for long-horizon manipulation with geometric and symbolic scene graphs," *arXiv preprint arXiv:2012.07277*, 2020.

[8] M. Fox and D. Long, "Pddl2. 1: An extension to pddl for expressing temporal planning domains," *Journal of artificial intelligence research*, vol. 20, pp. 61–124, 2003.

[9] B. Kim and L. Shimanuki, "Learning value functions with relational state representations for guiding task-and-motion planning," in *Conference on Robot Learning*. PMLR, 2020, pp. 955–968.

[10] D. Driess, J.-S. Ha, and M. Toussaint, "Deep visual reasoning: Learning to predict action sequences for task and motion planning from an initial scene image," *arXiv preprint arXiv:2006.05398*, 2020.

[11] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 5–33, 2001.

[12] E. Gizzi, M. G. Castro, and J. Sinapov, "Creative problem solving by robots using action primitive discovery," in *International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*. IEEE, 2019, pp. 228–233.

[13] V. Sarathy, D. Kasenberg, S. Goel, J. Sinapov, and M. Scheutz, "Spotter: Extending symbolic planning operators through targeted reinforcement learning," *arXiv preprint arXiv:2012.13037*, 2020.

[14] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "PDDLStream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2020.

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

[16] R. Chitnis, D. Hadfield-Menell, A. Gupta, S. Srivastava, E. Groshev, C. Lin, and P. Abbeel, "Guided search for task and motion plans using learned heuristics," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 447–454.

[17] B. Kim, L. P. Kaelbling, and T. Lozano-Pérez, "Learning to guide task and motion planning using score-space representation," in *International Conference on Robotics and Automation (ICRA)*, 2017.

[18] R. Chitnis, L. P. Kaelbling, and T. Lozano-Pérez, "Learning quickly to plan quickly using modular meta-learning," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2019.

[19] A. Arora, H. Fiorino, D. Pellier, M. Métivier, and S. Pesty, "A review of learning planning action models," *The Knowledge Engineering Review*, vol. 33, 2018.

[20] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling, "Learning symbolic models of stochastic domains," *Journal of Artificial Intelligence Research*, vol. 29, pp. 309–352, 2007.

[21] C. Rodrigues, P. Gérard, C. Rouveirol, and H. Soldano, "Active learning of relational action models," in *International Conference on Inductive Logic Programming*. Springer, 2011, pp. 302–316.

[22] S. N. Cresswell, T. L. McCluskey, and M. M. West, "Acquiring planning domain models using LOCM," *The Knowledge Engineering Review*, vol. 28, no. 2, pp. 195–213, 2013.

[23] D. Aineto, S. Jiménez, and E. Onaindia, "Learning STRIPS action models with classical planning," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2018.

[24] V. Xia, Z. Wang, K. R. Allen, T. Silver, and L. P. Kaelbling, "Learning sparse relational transition models," in *ICLR*, 2019.

[25] B. Marthi, S. J. Russell, and J. A. Wolfe, "Angelic semantics for high-level actions." in *ICAPS*, 2007, pp. 232–239.

[26] H. L. Younes and M. L. Littman, "PPDDL1.0: The language for the probabilistic part of IPC-4," in *IPC*, 2004.

[27] S. Muggleton, "Inductive logic programming," *New generation computing*, vol. 8, no. 4, pp. 295–318, 1991.

[28] S. W. Yoon, A. Fern, and R. Givan, "FF-Replan: A baseline for probabilistic planning." in *ICAPS*, vol. 7, 2007, pp. 352–359.

[29] E. Coumans and Y. Bai, "PyBullet, a python module for physics simulation for games, robotics and machine learning," *GitHub*, 2016.

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