Inventing Relational State and Action Abstractions for Effective and Efficient Bilevel Planning

Tom Silver*, **Rohan Chitnis***, **Nishanth Kumar**, **Willie McClinton**, **Tomás Lozano-Pérez**, **Leslie Pack Kaelbling** and **Joshua Tenenbaum**
MIT Computer Science and Artificial Intelligence Laboratory
{tslvr, ronuchit, njk, wbm3, tlp, lpk, jbt}@mit.edu

Abstract

Effective and efficient planning in continuous state and action spaces is fundamentally hard, even when the transition model is deterministic and known. One way to alleviate this challenge is to perform bilevel planning with abstractions, where a high-level search for abstract plans is used to guide planning in the original transition space. In this paper, we learn state and action abstractions that are explicitly optimized for both effective (successful) and efficient (fast) bilevel planning. Given demonstrations, our data-efficient approach learns relational, neuro-symbolic abstractions that generalize over object identities and numbers. The symbolic components resemble the STRIPS operators found in AI planning, and the neural components refine the abstractions into executable actions. Experimentally, we show across four robotic planning environments that our learned abstractions are able to quickly solve held-out tasks of longer horizons than were seen in the demonstrations, and outperform the efficiency of abstractions that we manually specified. We also find that as the planner configuration varies, the learned abstractions adapt accordingly, indicating that our abstraction learning method is both "task-aware" and "planner-aware."

1 Introduction

An autonomous agent should make *good* decisions *quickly*. These two considerations — effectiveness and efficiency — are especially important, and often competing, when an agent is *planning* in long-horizon, continuous-space tasks. *Abstractions* offer a mechanism to overcome this intractability [Li *et al.*, 2006; Konidaris *et al.*, 2018]. While state and action abstractions have a rich history in AI and robotics, a major limitation of early work is the downward refinability assumption [Marthi *et al.*, 2007]: that planning can be decomposed into first searching for an abstract plan, and then refining it into an actual plan. This is untenable in many applications, especially in robotics, where complex geometric constraints cannot be easily abstracted. To avoid this assumption, we consider *bilevel planning*, where reasoning in a high-level abstraction provides guidance for reasoning in a low-level task [Garrett *et al.*, 2021]. Another clear limitation of early work on abstractions is the reliance on manual specification, which requires understanding not only the environment, but also the interplay between the abstractions and the planner.

We identify three key desiderata of a system for planning with state and action abstractions:

1. These state and action abstractions should be learned, not manually designed for each environment.
2. Planning with the learned abstractions should be tolerant to violations of the downward refinability assumption.
3. The abstractions should be trained to explicitly optimize both the effectiveness and the efficiency of planning.

In this paper, we develop a framework for learning abstractions for planning that addresses all three desiderata. Specifically, *we learn state and action abstractions that are explicitly optimized for effective and efficient bilevel planning*. We consider learning from a modest number of demonstrations (around 50-200 per environment in our experiments) in deterministic, fully observed, goal-based planning problems. The problems have object-centric continuous states and hybrid discrete-continuous actions, as are common in robotics [Garrett *et al.*, 2021]. To obtain data-efficient generalization over object identities, we learn *relational*, *neuro-symbolic* abstractions, where the symbolic components are predicates and operators, like those used in AI planning [Fikes and Nilsson, 1971], and the neural components are samplers that refine the abstractions into actions that can be executed in the actual environment [Chitnis *et al.*, 2016; Kim *et al.*, 2018].

In experiments across four robotic planning environments, we find that our framework is very data-efficient, and that the resulting learned abstractions are both "task-aware" and "planner-aware." We demonstrate task-awareness by evaluating the learned abstractions in held-out tasks involving different numbers of objects, longer horizons, and larger goal expressions than were seen in the demonstrations, finding them to lead to effective and efficient planning. Interestingly, we find that in some environments, the learned abstractions can even outperform ones that we manually specified. For planner-awareness, we show that as the configuration of the planner varies, the learned abstractions adapt accordingly. We compare against several baselines and ablations of our system to further validate our results.

*First two authors contributed equally.

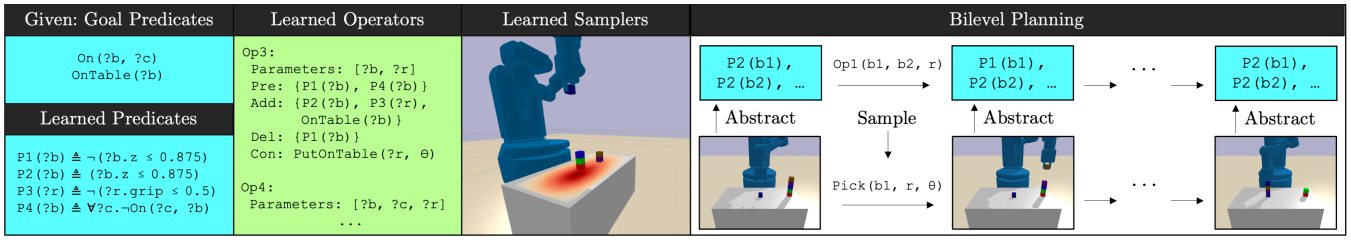


Figure 1: **Overview of our framework.** Given a small set of goal predicates (first panel, top), we use demonstration data to learn new predicates (first panel, bottom). In this Blocks example, implemented using the PyBullet physics simulator [Coumans and Bai, 2016], the learned predicates `P1` – `P4` intuitively represent `Holding`, `NotHolding`, `HandEmpty`, and `NothingAbove` respectively. Collectively, the predicates define a state abstraction that maps continuous states x in the environment to abstract states s. Object types are omitted for clarity. After predicate invention, we learn abstractions of the continuous action space and transition model via planning operators (second panel). For each operator, we learn a sampler (third panel), a neural network that maps continuous object features in a given state to continuous action parameters for controllers which can be executed in the environment. In this example, the sampler proposes different placements on the table for the held block. With these learned representations, we perform bilevel planning (fourth panel), with search in the abstract spaces guiding planning in the continuous spaces. Here, the goal is to create two specific towers of blocks.

2 Problem Setting

We consider learning from demonstrations in deterministic planning problems. These problems are goal-based and object-centric, with continuous states and hybrid discrete-continuous actions. Formally, an *environment* is a tuple $\langle \Lambda, d, \mathcal{C}, f, \Psi_G \rangle$, and is associated with a distribution $\mathcal{T}$ over *tasks*, where each task $T \in \mathcal{T}$ is a tuple $\langle \mathcal{O}, x_0, g \rangle$.

Λ is a finite set of object *types*, and the map $d : \Lambda \to \mathbb{N}$ defines the dimensionality of the real-valued feature vector for each type. Within a task, $\mathcal{O}$ is an *object set*, where each object has a type drawn from Λ; this $\mathcal{O}$ can (and typically will) vary between tasks. The $\mathcal{O}$ induces a state space $\mathcal{X}_\mathcal{O}$ (going forward, we simply write $\mathcal{X}$ when clear from context). A *state* $x \in \mathcal{X}$ in a task is a mapping from each $o \in \mathcal{O}$ to a feature vector in $\mathbb{R}^{d(\text{type}(o))}$; x_0 is the initial state of the task.

$\mathcal{C}$ is a finite set of *controllers*. A controller $C((\lambda_1, \ldots, \lambda_v), \Theta) \in \mathcal{C}$ can have both discrete typed parameters $(\lambda_1, \ldots, \lambda_v)$ and a continuous real-valued vector of parameters Θ. For instance, a controller `Pick` for picking up a block might have one discrete parameter of type `block` and a Θ that is a placeholder for a specific grasp pose. The controller set $\mathcal{C}$ and object set $\mathcal{O}$ induce an action space $\mathcal{A}_\mathcal{O}$ (going forward, we write $\mathcal{A}$ when clear). An *action* $a \in \mathcal{A}$ in a task is a controller $C \in \mathcal{C}$ with both discrete and continuous arguments: $a = C((o_1, \ldots o_v), \theta)$, where the objects $(o_1, \ldots o_v)$ are drawn from the object set $\mathcal{O}$ and must have types matching the controller's discrete parameters $(\lambda_1, \ldots, \lambda_v)$. Transitions through states and actions are governed by $f : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$, a known, deterministic transition model that is shared across tasks.

A *predicate* ψ is characterized by an ordered list of types $(\lambda_1, \ldots, \lambda_m)$ and a lifted binary state classifier $c_\psi : \mathcal{X} \times \mathcal{O}^m \to \{\text{true}, \text{false}\}$, where $c_\psi(x, (o_1, \ldots, o_m))$ is defined only when each object o_i has type λ_i. For instance, the predicate `Holding` may, given a state and two objects, robot and block, describe whether the block is held by the robot in this state. A *lifted atom* is a predicate with typed variables (e.g., `Holding(?robot, ?block)`). A *ground atom* $\underline{\psi}$ consists of a predicate ψ and objects $(o_1, \ldots, o_m)$, again with all $\text{type}(o_i) = \lambda_i$ (e.g., `Holding(robby, block7)`). Note that a ground atom induces a binary state classifier $c_{\underline{\psi}} : \mathcal{X} \to \{\text{true}, \text{false}\}$, where $c_{\underline{\psi}}(x) \triangleq c_\psi(x, (o_1, \ldots, o_m))$.

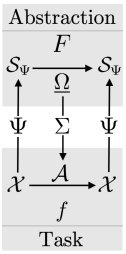
Ψ_G is a small set of *goal predicates* that we assume are given and sufficient for representing task goals, but insufficient practically as standalone state abstractions. Specifically, the goal g of a task is a set of ground atoms over predicates in Ψ_G and objects in $\mathcal{O}$. A goal g is said to *hold* in a state x if for all ground atoms $\underline{\psi} \in g$, the classifier $c_{\underline{\psi}}(x)$ returns true. A solution to a task is a *plan* $\pi = (a_1, \ldots, a_n)$, a sequence of actions $a \in \mathcal{A}$ such that successive application of the transition model $x_i = f(x_{i-1}, a_i)$ on each $a_i \in \pi$, starting from initial state x_0, results in a final state x_n where g holds.

The agent is provided with a set of *training tasks* from $\mathcal{T}$ and a set of demonstrations $\mathcal{D}$, with *one demonstration per task*. We assume action costs are unitary and demonstrations are near-optimal, which will be exploited in Section 5. Each demonstration consists of a training task $\langle \mathcal{O}, x_0, g \rangle$ and a plan π^* that solves the task. Note that for each π^*, we can recover the associated state sequence starting at x_0, since f is known and deterministic. The agent's objective is to *efficiently* solve held-out tasks drawn from $\mathcal{T}$, using anything it chooses to learn from the demonstrations. In other words, the agent should produce *good* solutions *quickly*. In our experiments, to assess generalization, we evaluate the agent on tasks that involve more objects, require longer action sequences, and have larger goal expressions than the training tasks.

3 Key Representations

Since the agent has access to the transition model f, one approach for optimizing the objective described in Section 2 is to forgo learning entirely and plan over the state state $\mathcal{X}$ and action space $\mathcal{A}$. However, planning directly in these large spaces is highly infeasible. Instead, we propose to *learn abstractions* using demonstrations. We adopt a very general definition of an abstraction [Konidaris and Barto, 2009]: mappings from $\mathcal{X}$ and $\mathcal{A}$ to alternative state and action spaces. In this section, we describe representations; in Section 4, we discuss planning; and in Section 5, we address learning.

We first characterize an abstract state space $\mathcal{S}_\Psi$ and a transformation from states in $\mathcal{X}$ to abstract states. Next, we describe an abstract action space $\underline{\Omega}$ and an abstract transition model $F : \mathcal{S}_\Psi \times \underline{\Omega} \to \mathcal{S}_\Psi$ that can be used to plan in the abstract space. Finally, we define samplers Σ for refining abstract actions back into $\mathcal{A}$, so that abstract plans can guide planning in the task. See the diagram on the right for a summary.



(1) An abstract state space. We use a set of predicates Ψ to induce an abstract state space $\mathcal{S}_\Psi$. Recalling that a ground atom $\underline{\psi}$ induces a classifier $c_{\underline{\psi}}$ over states $x \in \mathcal{X}$, we have:

Definition 1 (Abstract state)**.** *An* abstract state s *is the set of ground atoms under* Ψ *that hold true in* x*:*

$$s = \textsc{Abstract}(x, \Psi) \triangleq \{\underline{\psi} : c_{\underline{\psi}}(x) = \text{true}, \forall \psi \in \Psi\}.$$

The (discrete) abstract state space induced by Ψ is denoted $\mathcal{S}_\Psi$. Throughout this work, we use predicate sets Ψ that are supersets of the given goal predicates Ψ_G. However, only the goal predicates are given, and they alone are typically very limited; in Section 5, we will discuss how the agent can use data to *invent predicates* that will make up the majority of Ψ. See Figure 1 (first panel) for an example of a predicate set Ψ, made up of goal predicates and learned predicates.

(2) An abstract action space and abstract transition model. We address both by having the agent learn *operators*:

Definition 2 (Operator)**.** *An* operator *is a tuple* $\omega = \langle \textsc{Par}, \textsc{Pre}, \textsc{Eff}^+, \textsc{Eff}^-, \textsc{Con} \rangle$ *where:*

- Par *is an ordered list of* parameters*: variables with types drawn from the type set* Λ*.*
- $\textsc{Pre}, \textsc{Eff}^+, \textsc{Eff}^-$ *are* preconditions*,* add effects*, and* delete effects*, each a set of lifted atoms over* Ψ *and* Par*.*
- Con *is a tuple* $\langle C, \textsc{Par}_{\textsc{Con}} \rangle$ *where* $C((\lambda_1, \ldots, \lambda_v), \Theta) \in \mathcal{C}$ *is a controller and* $\textsc{Par}_{\textsc{Con}}$ *is an ordered list of* controller arguments*, each a variable from* Par*. Furthermore,* $|\textsc{Par}_{\textsc{Con}}| = v$*, and each argument* i *must be of the respective type* λ_i*.*

We denote the set of operators as Ω. See Figure 1 (second panel) for an example. Unlike in STRIPS, our operators are augmented with controllers and controller arguments, which will help us connect to the task actions in **(3)** below. Now, given a task with object set $\mathcal{O}$, the set of all *ground operators* defines our (discrete) abstract action space for a task:

Definition 3 (Ground operator / abstract action)**.** *A* ground operator $\underline{\omega} = \langle \omega, \delta \rangle$ *is an operator* ω *and a substitution* $\delta : \textsc{Par} \to \mathcal{O}$ *mapping parameters to objects. We use* $\underline{\textsc{Pre}}, \underline{\textsc{Eff}^+}, \underline{\textsc{Eff}^-}$*, and* $\underline{\textsc{Par}_{\textsc{Con}}}$ *to denote the ground preconditions, ground add effects, ground delete effects, and ground controller arguments of* $\underline{\omega}$*, where variables in* Par *are substituted with objects under* δ*.*

We denote the set of ground operators (the abstract action space) as $\underline{\Omega}$. Together with the abstract state space $\mathcal{S}_\Psi$, the preconditions and effects of the operators induce an abstract transition model for a task:

Definition 4 (Abstract transition model)**.** *The* abstract transition model *induced by predicates* Ψ *and operators* Ω *is a partial function* $F : \mathcal{S}_\Psi \times \underline{\Omega} \to \mathcal{S}_\Psi$*.* $F(s, \underline{\omega})$ *is only defined if* $\underline{\omega}$ *is* applicable *in* s*:* $\underline{\textsc{Pre}} \subseteq s$*. If defined,* $F(s, \underline{\omega}) \triangleq (s - \underline{\textsc{Eff}^-}) \cup \underline{\textsc{Eff}^+}$.

(3) A mechanism for refining abstract actions into task actions. A ground operator $\underline{\omega}$ induces a partially specified controller, $C((o_1, \ldots o_v), \Theta)$ with $(o_1, \ldots o_v) = \underline{\textsc{Par}_{\textsc{Con}}}$, where object arguments have been selected but continuous parameters Θ have not. To *refine* this abstract action $\underline{\omega}$ into a task-level action $a = C((o_1, \ldots o_v), \theta)$, we use *samplers*:

Definition 5 (Sampler)**.** *Each operator* $\omega \in \Omega$ *is associated with a* sampler $\sigma : \mathcal{X} \times \mathcal{O}^{|\textsc{Par}|} \to \Delta(\Theta)$*, where* $\Delta(\Theta)$ *is the space of distributions over* Θ*, the continuous parameters of the operator's controller.*

Definition 6 (Ground sampler)**.** *For ground operator* $\underline{\omega} \in \underline{\Omega}$*, if* $\underline{\omega} = \langle \omega, \delta \rangle$ *and* σ *is the sampler for* ω*, then the* ground sampler *for* $\underline{\omega}$ *is a state-conditioned distribution* $\underline{\sigma} : \mathcal{X} \to \Delta(\Theta)$*, where* $\underline{\sigma}(x) \triangleq \sigma(x, \delta(\textsc{Par}))$.

We denote the set of samplers as Σ. See Figure 1 (third panel) for an example.

What connects the transition model f, abstract transition model F, and samplers Σ? While previous works enforce the downward refinability property [Marthi *et al.*, 2007; Pasula *et al.*, 2007; Konidaris *et al.*, 2018], it is important in robotics to be robust to violations of this property, since learned abstractions will typically lose critical geometric information. Therefore, we only require our learned abstractions to satisfy the following *weak semantics*: for every ground operator $\underline{\omega}$ with partially specified controller $C((o_1, \ldots, o_v), \Theta)$ and associated ground sampler $\underline{\sigma}$, there exists some $x \in \mathcal{X}$ and some θ in the support of $\underline{\sigma}(x)$ such that $F(s, \underline{\omega})$ is defined and equals s', where $s = \textsc{Abstract}(x, \Psi)$, $a = C((o_1, \ldots, o_v), \theta)$, and $s' = \textsc{Abstract}(f(x, a), \Psi)$. Note that downward refinability [Marthi *et al.*, 2007] makes a much stronger assumption: that this statement holds for *every* $x \in \mathcal{X}$ where $F(s, \underline{\omega})$ is defined.

4 Bilevel Planning

To use the components of an abstraction — predicates Ψ, operators Ω, and samplers Σ — for efficient planning, we build on *bilevel* planning techniques [Garrett *et al.*, 2021; Srivastava *et al.*, 2014]. We conduct an outer search over *abstract plans* using the predicates and operators, and an inner search over refinements of an abstract plan into a task solution π using the predicates and samplers.

Definition 7 (Abstract plan)**.** *An* abstract plan $\hat{\pi}$ *for a task* $\langle \mathcal{O}, x_0, g \rangle$ *is a sequence of ground operators* $(\underline{\omega}_1, \ldots, \underline{\omega}_n)$ *such that applying the abstract transition model* $s_i = F(s_{i-1}, \underline{\omega}_i)$ *successively starting from* $s_0 = \textsc{Abstract}(x_0, \Psi)$ *results in a sequence of abstract states* $(s_0, \ldots, s_n)$ *that achieves the goal, i.e.,* $g \subseteq s_n$*. This* $(s_0, \ldots, s_n)$ *is called the* expected abstract state sequence*.*

Because downward refinability does not hold in our setting, an abstract plan $\hat{\pi}$ is *not* guaranteed to be refinable into a solution π for the task, which necessitates bilevel planning. We now describe the planning algorithm in detail.

PLAN(x_0, g, Ψ, Ω, Σ)
// Parameters: n_{abstract}, n_{samples}.
1 $s_0 \leftarrow$ ABSTRACT(x_0, Ψ)
2 **for** $\hat{\pi}$ in GENABSTRACTPLAN$(s_0, g, \Omega, n_{\text{abstract}})$
3 **if** REFINE$(\hat{\pi}, x_0, \Psi, \Sigma, n_{\text{samples}})$ succeeds w/ π
4 **return** π

Algorithm 1: Pseudocode for our bilevel planning algorithm. The inputs are an initial state x_0, goal g, predicates Ψ, operators Ω, and samplers Σ; the output is a plan π. An outer loop runs GENABSTRACTPLAN, which generates plans in the abstract state and action spaces. An inner loop runs REFINE, which attempts to concretize each abstract plan $\hat{\pi}$ into a plan π. If REFINE succeeds, then the found plan π is returned as the solution; if REFINE fails, then GENABSTRACTPLAN continues.

4.1 Algorithm Description

The overall structure of the planner is outlined in Algorithm 1. For the outer search that finds abstract plans $\hat{\pi}$, denoted GENABSTRACTPLAN (Alg. 1, Line 2), we leverage the STRIPS-style operators and predicates [Fikes and Nilsson, 1971] to automatically derive a domain-independent heuristic popularized by the AI planning community, such as LM-Cut [Helmert and Domshlak, 2009]. We use this heuristic to run an A* search over the abstract state space $\mathcal{S}_\Psi$ and abstract action space $\underline{\Omega}$. This A* search is used as a generator (hence the name GENABSTRACTPLAN) of abstract plans $\hat{\pi}$, outputting one at a time[1]. Parameter n_{abstract} governs the maximum number of abstract plans that can be generated before the planner terminates with failure.

For each abstract plan $\hat{\pi}$, we conduct an inner search that attempts to REFINE (Alg. 1, Line 3) it into a solution π (a plan that achieves the goal under the transition model f). While various implementations of REFINE are possible [Chitnis *et al.*, 2016], we follow [Srivastava *et al.*, 2014] and perform a backtracking search over the abstract actions $\underline{\omega}_i \in \hat{\pi}$. Recall that each $\underline{\omega}_i$ induces a partially specified controller $C_i((o_1, \ldots, o_v)_i, \Theta_i)$ and has an associated ground sampler $\underline{\sigma}_i$. To begin the search, we initialize an indexing variable i to 1. On each step of search, we sample continuous parameters $\theta_i \sim \underline{\sigma}_i(x_{i-1})$, which fully specify an action $a_i = C_i((o_1, \ldots, o_v)_i, \theta_i)$. We then check whether $x_i = f(x_{i-1}, a_i)$ obeys the expected abstract state sequence, i.e., whether $s_i = \text{ABSTRACT}(x_i, \Psi)$. If so, we continue on to $i \leftarrow i + 1$. Otherwise, we repeat this step, sampling a new $\theta_i \sim \underline{\sigma}_i(x_{i-1})$. Parameter n_{samples} governs the maximum number of times we invoke the sampler for a single value of i before backtracking to $i \leftarrow i - 1$. REFINE succeeds if the goal g holds when $i = |\hat{\pi}|$, and fails when i backtracks to 0.

If REFINE succeeds given a candidate $\hat{\pi}$, the planner terminates with success (Alg. 1, Line 4) and returns the plan $\pi = (a_1, \ldots, a_{|\hat{\pi}|})$. Crucially, if REFINE fails, we continue with GENABSTRACTPLAN to generate the next candidate $\hat{\pi}$. In the taxonomy of task and motion planners (TAMP), this approach is in the "search-then-sample" category [Srivastava *et al.*, 2014; Dantam *et al.*, 2016; Garrett *et al.*, 2021]. As we have described it, this planner is *not* probabilistically complete, because abstract plans are not revisited. Furthermore, there is no propagation of information from the inner search to the outer search. Extensions to address these limitations exist [Chitnis *et al.*, 2016], but are not our focus in this work.

[1]This usage of A* search as a generator is related to the field of top-k planning [Riabov *et al.*, 2014; Katz *et al.*, 2018; Ren *et al.*, 2021]. We experimented with off-the-shelf top-k planners, but chose to use A* because it was faster in our domains. Note that it is used heavily in the learning loop (Section 5).

5 Learning Abstractions

We wish to learn predicates Ψ, operators Ω, and samplers Σ data-efficiently, using the demonstrations $\mathcal{D}$. Due to space restrictions, we refer the reader to our prior work for a description of operator and sampler learning [Chitnis *et al.*, 2021], and focus here on *predicate invention*.

Inspired by prior work [Bonet and Geffner, 2019; Loula *et al.*, 2019; Curtis *et al.*, 2021], we approach the predicate invention problem from a program synthesis perspective [Cropper and Muggleton, 2016]. First, we define a compact representation of an infinite space of predicates in the form of a *grammar*. We then enumerate a large pool of *candidate predicates* from this grammar, with simpler candidates enumerated first. Next, we perform a *local search* over subsets of candidates, with the aim of identifying a good final subset to use as Ψ. The crucial question in this step is: what *objective function* should we use to guide the search over candidate predicate sets? We begin with this last question.

Scoring a Candidate Predicate Set

Ultimately, we want to find a set of predicates Ψ that will lead to effective and efficient planning, after we use the predicates to learn operators Ω and samplers Σ. The real objective we want to minimize can be expressed as:

$$J_{\text{real}}(\Psi) \triangleq \mathbb{E}_{(\mathcal{O}, x_0, g) \sim \mathcal{T}}[\text{TIME}(\text{PLAN}(x_0, g, \Psi, \Omega, \Sigma))],$$

where Ω and Σ are learned using Ψ, PLAN is the algorithm described in Section 4, and TIME$(\cdot)$ measures the time that PLAN takes to find a solution[2]. However, we need an objective that can be used to guide a *search* over candidate predicate sets, meaning the objective must be evaluated many times. Unfortunately, J_{real} is far too expensive for this, due to two speed bottlenecks: sampler learning, which involves training several neural networks; and the repeated calls to REFINE from within PLAN, which each perform backtracking search over an abstract plan. To overcome this intractability, we propose to use a *proxy objective* J_{proxy}, one that is cheaper to evaluate than J_{real}, but that approximately preserves ordering, i.e., $J_{\text{proxy}}(\Psi) < J_{\text{proxy}}(\Psi') \iff J_{\text{real}}(\Psi) < J_{\text{real}}(\Psi')$.

Identifying a proxy objective that balances the trade-off between tractability and fidelity to J_{real} can be very challenging. In the course of our research, we considered many options inspired by prior work, including per-operator prediction error [Pasula *et al.*, 2007; Silver *et al.*, 2021], bisimulation [Bonet and Geffner, 2019; Curtis *et al.*, 2021], and inverse planning-based objectives [Paxton *et al.*, 2016;

[2]If no plan can be found (e.g., a task is infeasible under the abstraction), TIME would return a large constant representing a timeout.

Zhi-Xuan *et al.*, 2020], but found them all to be divergent from J_{real}, leading to poor performance (see the baselines in Section 6). Our main insight was based on the following observation: although sampler learning and REFINE are slow, operator learning and abstract search (on the training tasks) are both fast — operator learning takes time linear in the size of the dataset, and abstract search is guided by powerful AI planning heuristics. Therefore, we can use these to design a proxy objective J_{proxy} that mirrors J_{real}, but with cheap approximation schemes to avoid the two bottlenecks.

In particular, we will consider a proxy objective that *estimates* the time it would take to solve the training tasks under the abstraction induced by a candidate predicate set Ψ, without using samplers or doing refinement. Recalling that our dataset $\mathcal{D}$ has one demonstration π^* for each training task $\langle \mathcal{O}, x_0, g \rangle$, we propose the following proxy objective:

$$J_{\text{proxy}}(\Psi) \triangleq \frac{1}{|\mathcal{D}|} \sum_{(\mathcal{O}, x_0, g, \pi^*) \in \mathcal{D}} [\text{ETPT}(x_0, g, \Psi, \Omega, \pi^*)],$$

where ETPT abbreviates ESTIMATETOTALPLANNINGTIME (see Algorithm 2). There are three key points to note about J_{proxy}, in comparison to J_{real}: (1) it estimates the expectation in J_{real} using an average over the training tasks; (2) it estimates planning times using the demonstration π^* of each training task; (3) it does not rely on samplers Σ.

To estimate the total planning time, we perform the same A* abstract search described in Section 4.1, using the operators Ω learned from Ψ (Alg. 2, Line 4). In the process, we keep track of two quantities: $p_{\text{terminate-here}}$, which is a probability estimating whether PLAN with learned samplers would terminate with success on this step; and t_{expected}, which approximates the cumulative time elapsed of PLAN thus far. Both quantities are initialized to 0 (Alg. 2, Lines 2-3).

To update $p_{\text{terminate-here}}$ on each abstract plan (Alg. 2, Lines 5-6), we must estimate both whether PLAN would have terminated *before* this step, and whether PLAN would terminate *on* this step. For the former, we can use $(1 - p_{\text{terminate-here}})$. For the latter, since PLAN terminates only if REFINE succeeds, we use a function called ESTIMATEREFINEPROB to approximate the probability of successfully refining the given abstract plan, if we were to learn samplers Σ and then call REFINE. While various implementations are possible, we use a simple strategy that leverages the demonstration:

$$\text{ESTIMATEREFINEPROB}(\hat{\pi}, \pi^*) \triangleq (1-\epsilon)\epsilon^{|\text{COST}(\hat{\pi}) - \text{COST}(\pi^*)|}.$$

Here, $\epsilon > 0$ is a small constant (10^{-5} in our experiments), and $\text{COST}(\cdot)$ is in our case simply the number of actions in the plan, due to unitary costs. The intuition for this geometric distribution is as follows. Since the demonstration π^* is assumed to be near-optimal, an abstract plan $\hat{\pi}$ that is cheaper than π^* should look suspicious; if such a $\hat{\pi}$ were refinable, then the demonstrator would have likely used it to produced a better demonstration. If $\hat{\pi}$ is more expensive than π^*, then even though this abstraction would eventually produce a refinable abstract plan, it may take a long time for the outer loop of the planner, GENABSTRACTPLAN, to get to it (Section 4.1). We note that this scheme for estimating refinability is surprisingly minimal, in that it needs only the cost of each demonstration rather than its contents.

ESTIMATETOTALPLANNINGTIME(x_0, g, Ψ, Ω, π^*)

```
// Note: does not take in samplers!
// Parameters: n_abstract, t_upper.
1  s_0 ← ABSTRACT(x_0, Ψ)
2  p_terminate-here ← 0.0
3  t_expected ← 0.0
4  for π̂ in GENABSTRACTPLAN(s_0, g, Ω, n_abstract) do
5      p_refined ← ESTIMATEREFINEPROB(π̂, π*)
6      p_terminate-here ← (1 − p_terminate-here) · p_refined
7      t_iter ← ESTIMATETIME(π̂, x_0, Ψ, Ω)
8      t_expected ← t_expected + p_terminate-here · t_iter
9  t_expected ← t_expected + (1 − p_terminate-here) · t_upper
10 return t_expected
```

Algorithm 2: Pseudocode for our predicate invention proxy objective. The structure mimics that of Algorithm 1, with commonalities shown in blue. See Section 5 for details.

To update t_{expected} on each abstract plan (Alg. 2, Lines 7-8), we use a function called ESTIMATETIME to approximate the time spent on this abstract plan, and weight the result of this function by $p_{\text{terminate-here}}$. To implement ESTIMATETIME, we sum up estimates of the *abstract search time* and of the *refinement time*. Since we are running abstract search, we can exactly measure its time; however, we use the cumulative number of nodes created by the A* search so far as a processor-independent estimate. To estimate refinement time, recall that REFINE performs a backtracking search, and so over many calls to REFINE, the potentially several that fail will dominate the one or zero that succeed. Therefore, we estimate refinement time as a large constant (10^3 in our experiments) that captures the average cost of an exhaustive backtracking search. Note that even though this is a constant, the fact that it is multiplied by $p_{\text{terminate-here}}$ (Alg. 2, Line 8) means that its impact on the overall score will vary greatly.

Before finishing, we add a final term to t_{expected} (Alg. 2, Line 9) corresponding to the probability that PLAN would fail to refine *any* skeleton ($t_{\text{upper}} = 10^5$ in our experiments). Finally, we return t_{expected} as our estimated planning time for a single training task (Alg. 2, Line 10). The expression for J_{proxy} sums up these return values over all the training tasks.

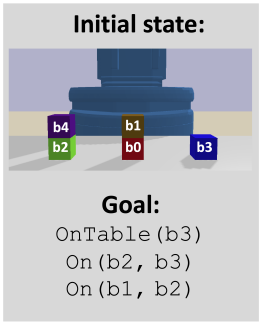
Local Search with the Proxy Objective

With our proxy objective J_{proxy} established, we turn to the question of how to best optimize it. We perform simple hill climbing, which has the benefit of being much more efficient than potential alternatives such as enforced hill climbing or greedy best-first search. We initialize search with the given goal predicates $\Psi_0 \leftarrow \Psi_G$, and add a single new predicate ψ from the candidate pool on each step i:

$$\Psi_{i+1} \leftarrow \underset{\psi \notin \Psi_i}{\text{argmin}}\, J_{\text{proxy}}(\Psi_i \cup \{\psi\}).$$

We repeat until no improvement can be found, and use the last predicate set as our final Ψ.

See Figure 2 for a real example of predicate invention via hill climbing search, taken from our experiments.



Search Iteration	Predicate Set	J_{proxy} (lower is better)	Success Rate on Evaluation Tasks
0	`On(?b, ?c)` `OnTable(?b)`	$1.3 \cdot 10^7$	0%

1	`On(?b, ?c)`
	`OnTable(?b)`
	$\neg$`(?b.z` $\leq$ `0.875)`
		$1.0 \cdot 10^7$	12%

2	`On(?b, ?c)`
	`OnTable(?b)`
	$\neg$`(?b.z` $\leq$ `0.875)`
	$\forall$`?c.`$\neg$`On(?c, ?b)`
		$2.0 \cdot 10^6$	14%

3	`On(?b, ?c)`
	`OnTable(?b)`
	$\neg$`(?b.z` $\leq$ `0.875)`
	$\forall$`?c.`$\neg$`On(?c, ?b)`
	$\neg$`(?r.grip` $\leq$ `0.5)`
		9351	100%

Abstract plans: Refinable?

```
[Stack b2 on b3, Stack b1 on b2]                                  ✗
                        ...
[Pick b2, Stack b2 on b3,                                         ✗
 Pick b1, Stack b1 on b2]
                        ...
[Pick b4, Pick b2, Stack b2 on b3,                                ✗
 Pick b1, Stack b1 on b2]
                        ...
[Pick b4, PutOnTable b4, Pick b2,                                 ✓
 Stack b2 on b3, Pick b1,
 Stack b1 on b2]
```

Figure 2: **Predicate invention via hill climbing**. (Left) An example task in Blocks. (Middle) Hill climbing over predicate sets, starting with the goal predicates Ψ_G. On each iteration, the single predicate that improves J_{proxy} the most is added to the set. The rightmost table column shows success rates under a 10-second timeout on a held-out set of evaluation tasks. (Right) Abstract plans generated by planning in the example task (left) with each predicate set (middle). Each iteration of hill climbing adds a predicate that causes all abstract plans above the dotted line to be pruned from consideration. At iteration 0, the robot believes it can achieve the goal by simply stacking `b2` on `b3` and `b1` on `b2`, even though it hasn't picked up either block. The first step of this abstract plan (red) is unrefinable. At iteration 1, a predicate with the intuitive meaning `Holding` is added, which makes the A* only consider abstract plans that pick up blocks before stacking them. Still, the abstract plan shown is unrefinable on the first step because `b4` is obstructing `b2`. At iteration 2, a predicate with the intuitive meaning `NothingAbove` is added, which leads the agent move `b4` out of the way before picking up `b2`. This plan is still unrefinable, though: the second step fails, because the abstraction still does not recognize that the robot cannot be holding two blocks simultaneously. Finally, at iteration 3, a predicate with the intuitive meaning `HandEmpty` is added, and the abstract search finds a refinable plan to solve the task.

Designing a Grammar of Predicates

Designing a grammar of predicates can in general be difficult, since there is a tradeoff between the expressivity of the grammar and the practicality of searching over it. For our experiments, we found that a simple grammar suffices:

- The base grammar includes two kinds of predicates: all the goal predicates Ψ_G, and *single-feature inequality classifiers*. These inequality classifiers are less-than-or-equal-to expressions that compare a constant against an individual feature dimension from $\{1, \ldots, d(\lambda)\}$, for some object type $\lambda \in \Lambda$. For the constant, we consider an infinite stream of numbers in the pattern $0.5, 0.25, 0.75, 0.125, 0.375, 0.625, 0.875, \ldots$, which represent *normalized* values of the feature, based on the range of values it takes on across all states in the dataset $\mathcal{D}$. We use this pattern because we want our grammar to describe an infinite stream of classifiers.
- Negations of predicates in the base grammar are included.
- We include two types of universal quantification: (1) quantifying over all variables, and (2) quantifying over all but one variable. An example of the first is `P() =` $\forall$ `?x, ?y . On(?x, ?y)`, while an example of the second is `P(?y) =` $\forall$ `?x . On(?x, ?y)`.
- Following prior work [Curtis *et al.*, 2021], we prune out candidate predicates if they are equivalent to any previously enumerated predicate, in terms of all groundings that hold in every state in the dataset $\mathcal{D}$.

See Figure 1 for examples of the kinds of predicates our grammar can generate. Note that there are many concepts this grammar cannot represent, but it is nevertheless rich enough to capture a wide class of state abstractions in practice. To generate our candidate predicate set for local search (Section 10), we enumerate n_{grammar} predicates from the grammar. In our experiments, we use $n_{\text{grammar}} = 200$.

6 Experiments

Our experiments are designed to answer the following questions: **(Q1)** To what extent do our learned abstractions help both the effectiveness and the efficiency of planning, and how do they compare to abstractions learned using other objective functions? **(Q2)** How do our learned state abstractions compare in performance to manually designed state abstractions? **(Q3)** How data-efficient is learning, with respect to the number of demonstrations? **(Q4)** Do our abstractions vary as we change the planner configuration, and if so, how?

6.1 Experimental Setup

We evaluate ten methods across four robotic planning environments. All experiments were conducted on a quad-core Intel Xeon Platinum 8260 processor, and all results are averaged over 10 random seeds, which vary the training and evaluation tasks, random initializations during learning, and tiebreaking during planning. For each seed, in all four environments, we sample a set of 50 *evaluation tasks* from the task distribution $\mathcal{T}$, with hyperparameters chosen to involve more objects and harder goals than were seen at training. Our key measures of *effective and efficient planning* are (1) success rate and (2) wall-clock time. Planning is limited to a 10-second timeout across all environments and methods.

Environments. Our first three environments were established in prior work [Silver *et al.*, 2021], but in that work, all state abstractions were manually defined (we use the same state abstractions for our Manual baseline below).

- **PickPlace1D.** In this toy environment, a robot must pick blocks and place them onto target regions along a table surface. All pick and place poses are in a 1D line. Evaluation tasks require 1-4 actions to solve.
- **Blocks.** In this environment, a robot in 3D must interact with blocks on a table to assemble them into towers. This

is a robotics adaptation of the blocks world domain in AI planning. Evaluation tasks require 2-20 actions to solve.

- **Painting.** In this challenging environment, a robot in 3D must pick, wash, dry, paint, and place widgets into either a box or a shelf, as specified by the goal. Evaluation tasks require 11-25 actions to solve.
- **Tools.** In this challenging environment, a robot operating on a 2D table surface must assemble contraptions by fastening screws, nails, and bolts, using a provided set of screwdrivers, hammers, and wrenches respectively. This environment has physical constraints outside the scope of our predicate grammar, and therefore tests the ability to cope with an insurmountable lack of downward refinability. Evaluation tasks require 7-20 actions to solve.

Methods. We evaluate our method, six baselines, a manually designed state abstraction, and two ablations.

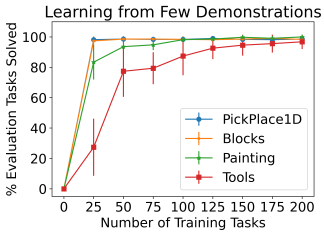
- **Ours.** Our main approach, which learns abstractions and uses them to guide planning on the evaluation tasks.
- **Bisimulation.** A baseline that learns abstractions by approximately optimizing the *bisimulation criteria* [Givan *et al.*, 2003], as in prior work [Curtis *et al.*, 2021]. Specifically, this baseline learns abstractions that minimize the number of transitions in the demonstrations where the abstract transition model F is applicable but makes a misprediction about the next abstract state.
- **Branching.** A baseline that learns abstractions by optimizing the *branching factor* of planning. Specifically, this baseline learns abstractions that aim to minimize the number of applicable operators in demonstration states.
- **Boltzmann.** A baseline that assumes the demonstrator is acting *noisily rationally* with respect to the cost-to-go under the (unknown) optimal abstractions. Specifically, for any candidate abstraction in our search, we compute the probability of the demonstration under a Boltzmann policy, with the AI planning heuristic used as a proxy for the true cost-to-go; we seek to maximize this probability.
- **GNN Shooting.** A baseline that trains a graph neural network [Battaglia *et al.*, 2018] policy. This GNN takes in the current state x, abstract state $s = \textsc{Abstract}(x, \Psi_G)$, and goal g. It outputs an action a, via a one-hot vector over $\mathcal{C}$ corresponding to which controller to execute, one-hot vectors over all objects at each discrete argument position, and a vector of continuous arguments. We train the GNN using behavior cloning on the data $\mathcal{D}$. At evaluation time, we sample trajectories by treating the outputted continuous arguments as the mean of a Gaussian with fixed variance. We use the known transition model f to check if the goal is achieved, and repeat until timeout.
- **GNN Model-Free.** A baseline that uses the same trained GNN as above, but directly executes the policy.
- **Random.** A baseline that simply executes a random controller with random arguments on each step. No learning.
- **Manual.** An oracle approach that plans with manually designed predicates for each environment.
- **Down Eval.** An ablation of Ours that uses $n_{\text{abstract}} = 1$ during evaluation only, in PLAN (Algorithm 1).
- **No Invent.** An ablation of Ours that uses $\Psi = \Psi_G$, i.e., only goal predicates are used for the state abstraction.

Additional details. All sampler neural networks are fully connected, with two hidden layers of size 32 each, and trained with the Adam optimizer [Kingma and Ba, 2014] for 1K epochs using learning rate 1e-3. The regressor networks are trained to predict a mean and covariance matrix of a multivariate Gaussian; this covariance matrix is restricted to be diagonal and PSD with an exponential linear unit [Clevert *et al.*, 2015]. For training the classifier networks, we subsample data to ensure a 1:1 balance between positive and negative examples. All AI planning heuristics are implemented using Pyperplan [Alkhazraji *et al.*, 2020]; all experiments use the LMCut heuristic unless otherwise specified. The planning parameters are $n_{\text{abstract}} = 1000$ for Tools and 8 for the other environments, and $n_{\text{samples}} = 1$ for Tools and 10 for the other environments.

6.2 Results and Discussion

Comparisons with baselines are shown in Figure 3, and allow us to answer **(Q1)**: our method solves many more held-out tasks within the timeout. A major reason for this performance gap is that unlike the baselines, our proxy objective J_{proxy} explicitly takes into account the effectiveness and efficiency of bilevel planning with candidate abstractions. Table 1 compares Ours with Manual and the two ablations. We can address **(Q2)** by comparing Ours to Manual in this table, which shows that the learned abstractions are on par with, and sometimes *better than*, our hand-designed abstractions.

Next, we look at the performance of the ablations in Table 1. The results for No Invent show that, as expected, using the goal predicates as a standalone state abstraction is completely insufficient for most tasks. Comparing Ours to Down Eval shows that assuming downward refinability at evaluation time works for PickPlace1D, Blocks, and Painting, but not for Tools. We were surprised by this result because the manually designed abstractions for PickPlace1D and Painting are *not* downward refinable [Silver *et al.*, 2021]. In contrast, the abstractions learned by Ours for the Tools environment are not downward refinable; for example, it is not possible to determine whether a screwdriver's shape is compatible with that of a screw, at the abstract level.



To address **(Q3)**, the figure on the right clearly shows the data efficiency of Ours. Each point shows a mean over 10 seeds, with standard deviations shown as vertical bars. Recall that we provide a single demonstration for each training task. In most environments, the figure shows that we obtain very good evaluation performance within just 50 demonstrations.

To address **(Q4)**, the table on the right shows an additional experiment we conducted in the Blocks environment, where we varied the AI planning heuristic used in predicate invention and evaluation. The Node column shows the number of nodes created during abstract search. While the gap in performance is limited with LMCut, our system shows a substantial improvement with hAdd. In the latter case, our approach invents four unary predicates with the intuitive meanings `Holding`, `NothingAbove`,

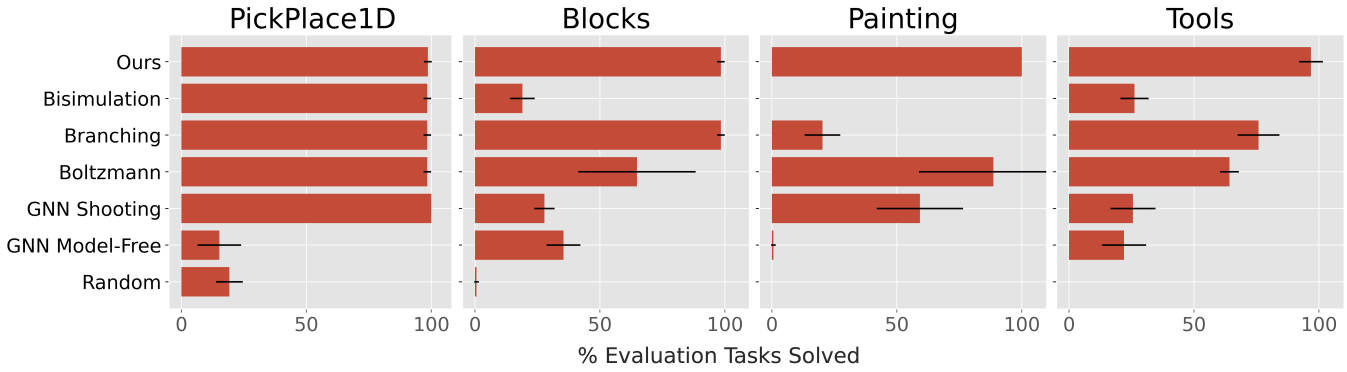


Figure 3: **Ours versus baselines.** Percentage of 50 evaluation tasks solved under a 10-second timeout, for all four environments. All results are averaged over 10 seeds. Horizontal black bars denote standard deviations.

	Ours			Manual			Down Eval			No Invent		
Environment	Succ	Node	Time	Succ	Node	Time	Succ	Node	Time	Succ	Node	Time
PickPlace1D	98.6	4.8	0.006	98.4	6.5	0.045	98.6	4.8	0.008	39.6	14.1	1.369
Blocks	98.4	2949	0.296	98.6	2941	0.251	98.2	2949	0.318	3.2	427.7	1.235
Painting	100.0	501.8	0.470	99.6	2608	0.464	98.8	489.0	0.208	0.0	–	–
Tools	96.8	1897	0.457	100.0	4771	0.491	42.8	152.5	0.060	0.0	–	–

Table 1: **Ours versus Manual and ablations.** Percentage of 50 evaluation tasks solved under a 10-second timeout (Succ), number of nodes created during GENABSTRACTPLAN (Node), and wall-clock planning time in seconds (Time). The latter two average over solved tasks only.

`HandEmpty`, and `NotOnAnyBlock`, to supplement the given goal predicates `On` and `OnTable`. Comparing these to Manual, which has the same predicates and operators as those in the International Planning Competition (IPC), we see the following differences: `Clear` is omitted, and `NothingAbove` and `NotOnAnyBlock` are added. We observed that the latter are logical transformations of predicates used in the standard IPC blocks world representation, which motivated us to run a separate, symbolic-only experiment, where we collected IPC blocks world problems and transformed them to use these predicates and associated learned operators. We found that using A* and hAdd, planning with our learned representations is much faster than planning with the IPC representations. For example, using Fast Downward [Helmert, 2006] on a problem from IPC 2000 with 36 blocks, planning succeeds with our representations in 12.5 seconds after approximately 7,000 expansions, whereas it fails within a 2 *hour* timeout with the standard encoding. These results are especially surprising because A* with hAdd is generally considered inferior to other heuristic search algorithms.

	Ours			Manual		
h	Succ	Node	Time	Succ	Node	Time
LMCut	98.4	2949	0.296	98.6	2941	0.251
hAdd	98.6	121.6	0.115	97.8	3883	0.235

7 Related Work

Our work continues a long line of research on learning state abstractions [Li *et al.*, 2006] and action abstractions [Arora *et al.*, 2018] for planning. Most relevant are works that learn symbolic state and action abstractions compatible with AI planners [Lang *et al.*, 2012; Jetchev *et al.*, 2013; Ugur and Piater, 2015; Asai and Fukunaga, 2018; Bonet and Geffner, 2019; Ahmetoglu *et al.*, 2020; Umili *et al.*, 2021]. Our work is particularly influenced by [Pasula *et al.*, 2007], who use search through a concept language to invent symbolic state and action abstractions, and [Konidaris *et al.*, 2018], who discover symbolic abstractions by leveraging the initiation and termination sets of a provided set of options that satisfy an abstract subgoal property.

Recent works have also considered learning abstractions for multi-level planning, like those in the task and motion planning (TAMP) [Garrett *et al.*, 2021] and hierarchical planning [Bercher *et al.*, 2019] literature. Some of these efforts consider learning symbolic action abstractions [Nguyen *et al.*, 2017; Silver *et al.*, 2021] or refinement strategies [Mandalika *et al.*, 2019; Chitnis *et al.*, 2021]; our operator and sampler learning methods take inspiration from these prior works. Recent efforts by Loula et al. (2019) and Curtis et al. (2021) consider learning both state and action abstractions for TAMP, like we do [Loula *et al.*, 2019; Curtis *et al.*, 2021]. The main distinguishing feature of our work is that our abstraction learning framework explicitly optimizes an objective that considers planning efficiency.

8 Conclusion

In this paper, we have described a framework for learning abstractions that are optimized for effective and efficient bilevel planning. In experiments, we learned relational, neuro-symbolic abstractions that generalize over object identities, efficiently solve long-horizon tasks, and even outperform manual abstractions. We also showed that these abstractions adapt to variations in the task and the planner.

References

[Ahmetoglu *et al.*, 2020] Alper Ahmetoglu, M Yunus Seker, Justus Piater, Erhan Oztop, and Emre Ugur. Deepsym: Deep symbol generation and rule learning from unsupervised continuous robot interaction for planning. *arXiv preprint arXiv:2012.02532*, 2020.

[Alkhazraji *et al.*, 2020] Yusra Alkhazraji, Matthias Frorath, Markus Grützner, Malte Helmert, Thomas Liebetraut, Robert Mattmüller, Manuela Ortlieb, Jendrik Seipp, Tobias Springenberg, Philip Stahl, and Jan Wülfing. Pyperplan, 2020.

[Arora *et al.*, 2018] Ankuj Arora, Humbert Fiorino, Damien Pellier, Marc Métivier, and Sylvie Pesty. A review of learning planning action models. *The Knowledge Engineering Review*, 33, 2018.

[Asai and Fukunaga, 2018] Masataro Asai and Alex Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *Proceedings of the aaai conference on artificial intelligence*, 2018.

[Battaglia *et al.*, 2018] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[Bercher *et al.*, 2019] Pascal Bercher, Ron Alford, and Daniel Höller. A survey on hierarchical planning-one abstract idea, many concrete realizations. In *IJCAI*, pages 6267–6275, 2019.

[Bonet and Geffner, 2019] Blai Bonet and Hector Geffner. Learning first-order symbolic representations for planning from the structure of the state space. *arXiv preprint arXiv:1909.05546*, 2019.

[Chitnis *et al.*, 2016] Rohan Chitnis, Dylan Hadfield-Menell, Abhishek Gupta, Siddharth Srivastava, Edward Groshev, Christopher Lin, and Pieter Abbeel. Guided search for task and motion plans using learned heuristics. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 447–454. IEEE, 2016.

[Chitnis *et al.*, 2021] Rohan Chitnis, Tom Silver, Joshua B Tenenbaum, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Learning neuro-symbolic relational transition models for bilevel planning. *arXiv preprint arXiv:2105.14074*, 2021.

[Clevert *et al.*, 2015] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[Coumans and Bai, 2016] Erwin Coumans and Yunfei Bai. PyBullet, a python module for physics simulation for games, robotics and machine learning. *GitHub repository*, 2016.

[Cropper and Muggleton, 2016] Andrew Cropper and Stephen H Muggleton. Learning higher-order logic programs through abstraction and invention. In *IJCAI*, pages 1418–1424, 2016.

[Curtis *et al.*, 2021] Aidan Curtis, Tom Silver, Joshua B Tenenbaum, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Discovering state and action abstractions for generalized task and motion planning. *arXiv preprint arXiv:2109.11082*, 2021.

[Dantam *et al.*, 2016] Neil T Dantam, Zachary K Kingston, Swarat Chaudhuri, and Lydia E Kavraki. Incremental task and motion planning: A constraint-based approach. In *Robotics: Science and systems*, volume 12, page 00052. Ann Arbor, MI, USA, 2016.

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

[Garrett *et al.*, 2021] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems*, 4:265–293, 2021.

[Givan *et al.*, 2003] Robert Givan, Thomas Dean, and Matthew Greig. Equivalence notions and model minimization in markov decision processes. *Artificial Intelligence*, 147(1-2):163–223, 2003.

[Helmert and Domshlak, 2009] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: what's the difference anyway? In *Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

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

[Jetchev *et al.*, 2013] Nikolay Jetchev, Tobias Lang, and Marc Toussaint. Learning grounded relational symbols from continuous data for abstract reasoning. In *Proceedings of the 2013 ICRA Workshop on Autonomous Learning*, 2013.

[Katz *et al.*, 2018] Michael Katz, Shirin Sohrabi, Octavian Udrea, and Dominik Winterer. A novel iterative approach to top-k planning. In *Proceedings of the Twenty-Eigth International Conference on Automated Planning and Scheduling (ICAPS 2018)*. AAAI Press, 2018.

[Kim *et al.*, 2018] Beomjoon Kim, Leslie Kaelbling, and Tomás Lozano-Pérez. Guiding search in continuous state-action spaces by learning an action sampler from off-target search experience. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[Kingma and Ba, 2014] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Konidaris and Barto, 2009] George Konidaris and Andrew Barto. Efficient skill learning using abstraction selection. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[Konidaris *et al.*, 2018] George Konidaris, Leslie Pack Kaelbling, and Tomas Lozano-Perez. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61:215–289, 2018.

[Lang *et al.*, 2012] Tobias Lang, Marc Toussaint, and Kristian Kersting. Exploration in relational domains for model-based reinforcement learning. *The Journal of Machine Learning Research*, 13(1):3725–3768, 2012.

[Li *et al.*, 2006] Lihong Li, Thomas J Walsh, and Michael L Littman. Towards a unified theory of state abstraction for MDPs. *ISAIM*, 4(5):9, 2006.

[Loula *et al.*, 2019] João Loula, Tom Silver, Kelsey R Allen, and Josh Tenenbaum. Discovering a symbolic planning language from continuous experience. In *Annual Meeting of the Cognitive Science Society (CogSci)*, page 2193, 2019.

[Mandalika *et al.*, 2019] Aditya Mandalika, Sanjiban Choudhury, Oren Salzman, and Siddhartha Srinivasa. Generalized lazy search for robot motion planning: Interleaving search and edge evaluation via event-based toggles. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 745–753, 2019.

[Marthi *et al.*, 2007] Bhaskara Marthi, Stuart J Russell, and Jason Andrew Wolfe. Angelic semantics for high-level actions. In *ICAPS*, pages 232–239, 2007.

[Nguyen *et al.*, 2017] Chanh Nguyen, Noah Reifsnyder, Sriram Gopalakrishnan, and Hector Munoz-Avila. Automated learning of hierarchical task networks for controlling minecraft agents. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 226–231. IEEE, 2017.

[Pasula *et al.*, 2007] Hanna M Pasula, Luke S Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

[Paxton *et al.*, 2016] Chris Paxton, Felix Jonathan, Marin Kobilarov, and Gregory D Hager. Do what i want, not what i did: Imitation of skills by planning sequences of actions. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3778–3785. IEEE, 2016.

[Ren *et al.*, 2021] Tianyu Ren, Georgia Chalvatzaki, and Jan Peters. Extended tree search for robot task and motion planning. *arXiv preprint arXiv:2103.05456*, 2021.

[Riabov *et al.*, 2014] Anton Riabov, Shirin Sohrabi, and Octavian Udrea. New algorithms for the top-k planning problem. In *Proceedings of the scheduling and planning applications workshop (spark) at the 24th international conference on automated planning and scheduling (icaps)*, pages 10–16, 2014.

[Silver *et al.*, 2021] Tom Silver, Rohan Chitnis, Joshua Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning symbolic operators for task and motion planning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3182–3189. IEEE, 2021.

[Srivastava *et al.*, 2014] Siddharth Srivastava, Eugene Fang, Lorenzo Riano, Rohan Chitnis, Stuart Russell, and Pieter Abbeel. Combined task and motion planning through an extensible planner-independent interface layer. In *2014 IEEE international conference on robotics and automation (ICRA)*, pages 639–646. IEEE, 2014.

[Ugur and Piater, 2015] Emre Ugur and Justus Piater. Bottom-up learning of object categories, action effects and logical rules: From continuous manipulative exploration to symbolic planning. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2627–2633. IEEE, 2015.

[Umili *et al.*, 2021] Elena Umili, Emanuele Antonioni, Francesco Riccio, Roberto Capobianco, Daniele Nardi, and Giuseppe De Giacomo. Learning a symbolic planning domain through the interaction with continuous environments. *ICAPS PRL Workshop*, 2021.

[Zhi-Xuan *et al.*, 2020] Tan Zhi-Xuan, Jordyn Mann, Tom Silver, Josh Tenenbaum, and Vikash Mansinghka. Online bayesian goal inference for boundedly rational planning agents. *Advances in Neural Information Processing Systems*, 33:19238–19250, 2020.