Learning State and Action Abstractions for Effective and Efficient Planning

by

Rohan Chitnis

B.S., University of California, Berkeley (2016)
S.M., Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 13, 2022

Certified by .....
Leslie P. Kaelbling
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Tomás Lozano-Pérez
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Learning State and Action Abstractions for Effective and Efficient Planning

by

Rohan Chitnis

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

An autonomous agent should make *good* decisions *quickly*. These two considerations — effectiveness and efficiency — are especially important, and often competing, when an agent *plans* to make decisions sequentially in long-horizon tasks. Unfortunately, planning directly in the state and action spaces of a task is intractable for many tasks of interest. *Abstractions* offer a mechanism for overcoming this intractability, allowing the agent to reason at a higher level about the most salient aspects of a task. In this thesis, we develop novel frameworks for learning state and action abstractions that are optimized for both effective and efficient planning. Most generally, state and action abstractions are arbitrary transformations of the state and action spaces of the given planning problem; we focus on *task-specific* abstractions that leverage the structure of a given task (or family of tasks) to make planning efficient. Throughout the chapters, we show how to learn neuro-symbolic abstractions for bilevel planning; present a method for learning to generate context-specific abstractions of Markov decision processes; formalize and give a tractable algorithm for reasoning efficiently about relevant exogenous processes in a Markov decision process; and introduce a powerful and general mechanism for planning in large problem instances containing many objects. We demonstrate across both classical and robotics planning tasks, using a wide variety of planners, that the methods we present optimize a tradeoff between planning effectively and planning efficiently.

Thesis Supervisor: Leslie P. Kaelbling
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Tomás Lozano-Pérez
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis is dedicated to several people who have been pivotal to the last six years of my life, as I went from understanding very little about myself and my research field, to understanding much more about myself and a bit more about my research field.

First and foremost, I'd like to thank my advisers Leslie Kaelbling and Tomás Lozano-Pérez. You both have been incredible advisers, and constantly gave me the freedom to explore my research passions without worrying about pressure to publish or work on the things that are considered cool right now. I really appreciate your support, both personal and professional, as I've grown over these last six years.

I'd also like to thank George Konidaris, the third member of my thesis committee, for many productive conversations over the years, and for helping shape this thesis into what it is. Your work has been inspirational to me, and crucial in shaping my own research.

To Tom Silver: you've been a wonderful collaborator and dear friend over the years. I am so thankful that we got fortuitously placed together on a project about curiosity many years ago; in retrospect, that was one of the defining moments of my PhD. I wish you nothing but success in continuing our research agenda, and in all your future endeavors. I also want to thank all my other collaborators over the years: Nishanth Kumar, Willie McClinton, Beomjoon Kim, Aidan Curtis, Ferran Alet, Kaiyu Zheng, Yoonchang Sung, Clement Gehring, Caelan Garrett, and Rachel Holladay. Thank you for giving me the opportunity to learn from and alongside you.

On the personal side, I want to thank all my friends, both from the Sidney-Pacific residence and in my department, for helping me have a community outside the lab.

I also want to thank my family friend Shirish Joshi for the great advice given to me during the first year of my program. I have never forgotten the words you spoke that day; I still think about them regularly, and they are a large part of the reason I was able to finish my PhD.

I want to finish by thanking my family. Thank you to my parents, Sunil and

Shubhada Chitnis, whose generosity and nonstop support has enabled me to work hard to get where I am today. My mom, especially, has sacrificed so much in her own life to enable me to succeed. Thank you to my extended family: Aji, Bobby, Sanju Mama, Aarti Mami, Shreyus, and Anish, for talking to me regularly and making sure that I'm doing okay. Sanju Mama: your PhD thesis from MIT many years ago was dedicated to me, in the hopes that I'd be inspired to pursue "a lifetime of learning" — I really appreciated that, and my thesis is dedicated to Shreyus and Anish doing the same. Finally, to Shalini Gupta, you've been the greatest partner I could ever hope for, through both the good times and the hard times these past few years. I'm so lucky to have met you, and I'm so happy we got the chance to take this journey through the PhD together. Looking forward to what's in store ahead.

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Algorithms

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 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* to make decisions sequentially in long-horizon tasks [96]. In order for robots to be able to support us in a wide range of activities in our everyday lives, such as cooking, cleaning, and running errands, this sort of planning is essential: the robot must be able to reason about its future behavior, as humans do very naturally. It must think about the effects of its actions and figure out how to chain together its various capabilities (e.g., moving an end effector or performing a detection) to produce effective sequences of actions that drive it to complete the tasks given to it.

Unfortunately, planning is hard — and often intractable — for many settings of interest, spanning discrete symbolic reasoning [66], stochastic shortest paths [124], and robotics domains with continuous state and action spaces [52]. A major issue is that planners are highly sensitive to the choice of state and/or action representation, but these representations are usually hand-specified for the entire domain by a designer, rather than tailored to any particular task (or family of tasks). This is unsatisfying because hand-specified representations cannot scale effectively, and limit the potential of designing autonomous systems for a variety of applications.

This thesis explores the idea of learning *abstractions* as a general mechanism for ameliorating the intractability of planning. Abstractions, in the most general

sense, allow the agent to reason at a higher level about the most salient aspects of a task [42, 44, 98, 91, 1]. Importantly, the learned abstractions should be specific to the tasks the agent will need to solve: they should simplify the representation of the task while preserving the aspects that are important for generating a good solution. We will explore two forms of task-specific abstractions, with a common goal of *using an abstraction for effective and efficient planning*. The first form, studied in Chapter 3, is *neuro-symbolic, relational* abstractions, which allow a robot to plan to long horizons in continuous spaces by learning STRIPS-style predicates and operators [51] and neural network samplers, then using them for bilevel hierarchical planning. The second form, studied in Chapters 4, 5, and 6, is *projective* abstractions, which detect and drop irrelevant variables from a factored planning problem to make it easier to solve.

Throughout this thesis, we will focus on *state abstractions* and *action abstractions*. An abstraction over state space $\mathcal{S}$ and action space $\mathcal{A}$ is a pair of functions (σ, τ), with $\sigma : \mathcal{S} \mapsto \mathcal{S}'$ and $\tau : \mathcal{A} \mapsto \mathcal{A}'$, where $\mathcal{S}'$ and $\mathcal{A}'$ are the abstract state and action spaces [91]. In words, this simply says that the abstraction maps states and actions into some other representation. Generally, we will be interested in abstractions that map to representations compatible with provided planners, such as classical AI planners, MDP solvers, or task and motion planners [52].

A central theme in this thesis will be the idea of optimizing (approximately) both the *effectiveness* of planning (the ability for a planner to return good solutions), and the *efficiency* of planning (the speed at which the planner returns its solutions). On one extreme, running a brute-force planner directly on a given task without suitable abstractions can produce solutions given enough time, but will be unusable in practice. On the other extreme, a planner that immediately returns bad solutions would be highly undesirable. It will often be the case that the methods presented in this thesis offer a good balance between these two criteria: we aim to learn task-specific abstractions that are not necessarily quality-preserving, but allow for faster planning without compromising too much quality.

Throughout the thesis, we will study algorithms for speeding up various planning

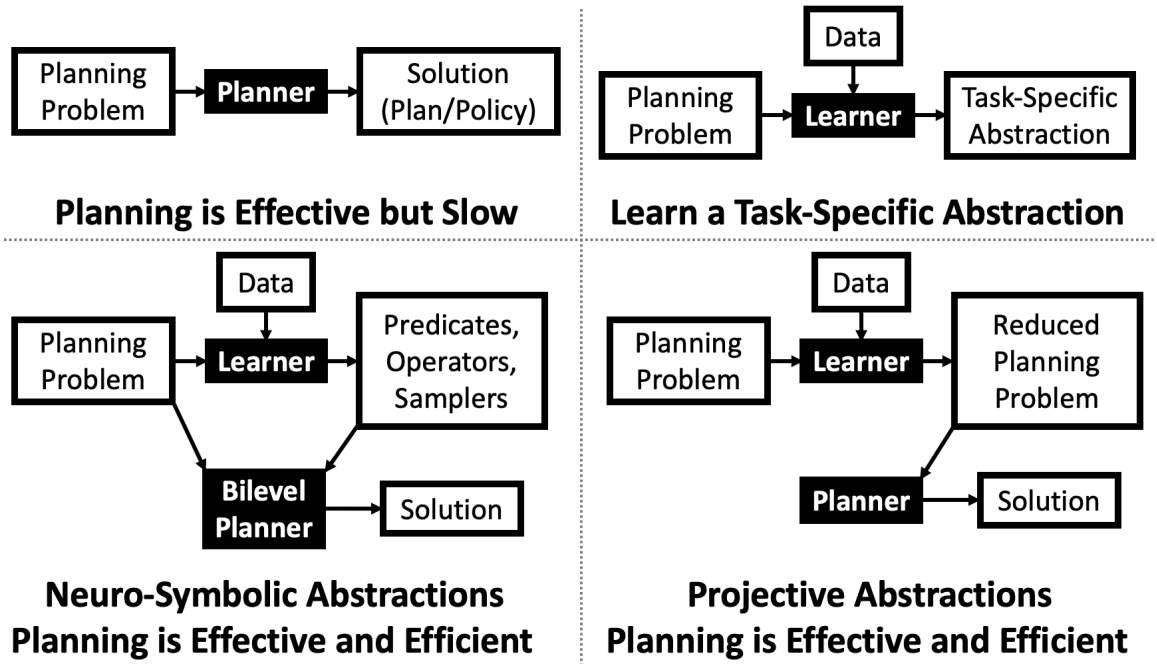


Figure 1-1: *Top Left*: We focus on solving large, continuous-space planning problems *effectively* and *efficiently*. Using a planner directly on large, complex problems can yield accurate solutions given enough time, but will be intractable in practice. *Top Right*: To remedy this intractability, we focus on learning task-specific abstractions from data. These abstractions can then be used to accelerate planning. *Bottom Left*: Chapter 3 of this thesis studies neuro-symbolic abstractions for bilevel planning. Here, we learn STRIPS-style symbolic predicates and operators, and neural network samplers, then use these for efficient hierarchical planning in both the abstraction and the original planning problem. *Bottom Right*: Chapters 4, 5, and 6 of this thesis study projective abstractions for factored planning problems. Here, we learn which variables can be dropped from consideration during planning, leading to a reduced problem that is more efficient to plan in than the original planning problem.

strategies, including task and motion planning, MDP value iteration, Monte Carlo tree search, and classical AI planning. We will also study a variety of planning problems, including geometry-aware robotics tasks and both deterministic and stochastic PDDL problems [107].

See Figure 1-1 for an overview diagram.

The remainder of this thesis is organized as follows:

- We begin by providing relevant background information in Chapter 2.
- In Chapter 3, we study how to invent STRIPS-style relational abstractions and

use them for planning in continuous spaces.

- In Chapter 4, we study a version of projective abstraction in which a robot imposes constraints *on its own behavior* in an environment to make some aspects of the environment irrelevant. We examine and solve two learning problems: which constraints to impose given a task, and what aspects of the environment become irrelevant under a given constraint.

- In Chapter 5, we study a version of projective abstraction in which the planning problem contains *exogenous processes*, which are processes that are unaffected by the robot's actions. We show how to leverage this property to design an efficient approximate algorithm for deciding which exogenous processes the planner should reason about.

- In Chapter 6, we study a version of projective abstraction in which the planning problem is *object-centric* and has *many objects*. Here, the robot must reason about which objects are relevant to solving any particular problem. We will show how to train graph neural networks [132] to perform this type of reasoning.

- Finally, we offer concluding thoughts and ideas for future research directions in Chapter 7.

1.1 Papers Covered in this Thesis

This thesis contains the contents of four papers, three of which are published at peer-reviewed conferences, and one of which is under review. Furthermore, all the work described here builds off several other of my previous works [29, 32, 33, 137].

Chapter 3 covers the following paper:

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, Joshua Tenenbaum. Under review. Preliminary version appeared at RLDM 2022.

Chapter 4 covers the following paper:

CAMPs: Learning Context-Specific Abstractions for Efficient Planning in Factored MDPs. **Rohan Chitnis***, Tom Silver*, Beomjoon Kim, Leslie Pack Kaelbling, Tomás Lozano-Pérez. In the proceedings of the Conference on Robot Learning (CoRL), 2020.

Chapter 5 covers the following paper:

Learning Compact Models for Planning with Exogenous Processes. **Rohan Chitnis**, Tomás Lozano-Pérez. In the proceedings of the Conference on Robot Learning (CoRL), 2019.

Chapter 6 covers the following paper:

Planning with Learned Object Importance in Large Problem Instances using Graph Neural Networks. Tom Silver*, **Rohan Chitnis***, Aidan Curtis, Joshua Tenenbaum, Tomás Lozano-Pérez, Leslie Pack Kaelbling. In the proceedings of the AAAI Conference on Artificial Intelligence (AAAI), 2021.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Background

In this thesis, we will be interested in decision-making problems where a robot interacts with an environment, taking actions and receiving observations of the resulting next states under some system dynamics. There are many possible ways to formalize such a setting. This section describes two popular frameworks: Markov decision processes and relational planning. In the former, the robot's objective is to maximize rewards accrued over an episode of interaction with the dynamical system (i.e., the environment). In the latter, the robot's objective is to achieve given goals starting from given initial states, where states and goals are represented in relational first-order logic. Both formulations can support either deterministic or stochastic planning; various parts of the thesis will study one or the other, and we will make it clear when a particular part is only applicable to the deterministic setting.

2.1 Markov Decision Processes and Factored Markov Decision Processes

An infinite-horizon Markov decision process (MDP) [124] is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$ where: $\mathcal{S}$ is the state space; $\mathcal{A}$ is the action space; $T(s_t, a_t, s_{t+1}) = P(s_{t+1} \mid s_t, a_t)$ is the state transition distribution with $s_t, s_{t+1} \in \mathcal{S}$ and $a_t \in \mathcal{A}$; $R(s_t, a_t)$ is the reward function; and γ is the discount factor in $(0, 1]$. On each timestep, the agent chooses an

action, transitions to a new state sampled from T, and receives a reward as specified by R. The solution to an MDP is a *policy* π, a mapping from states in $\mathcal{S}$ to actions in $\mathcal{A}$, such that the expected discounted sum of rewards over trajectories resulting from following π, which is $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t))\right]$, is maximized. Here, the expectation is with respect to the stochasticity in the initial state and state transitions. The *value* of a state $s \in \mathcal{S}$ under policy π is defined as the expected discounted sum of rewards from following π, starting at state s: $V_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)) \mid s_0 = s\right]$. The *value function* for π is the mapping from $\mathcal{S}$ to $\mathbb{R}$ defined by $V_\pi(s), \forall s \in \mathcal{S}$.

A finite-horizon MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, T, R, H \rangle$ is similar to an infinite-horizon one, but rather than a discount factor γ, a horizon H is given. The solution is a policy π that maximizes the expected sum of rewards up to the horizon: $\mathbb{E}\left[\sum_{t=0}^{H} R(s_t, \pi(s_t))\right]$. Dynamic programming algorithms (e.g., value iteration) are a common strategy for solving MDPs [124] by leveraging a recursive formulation of the value function.

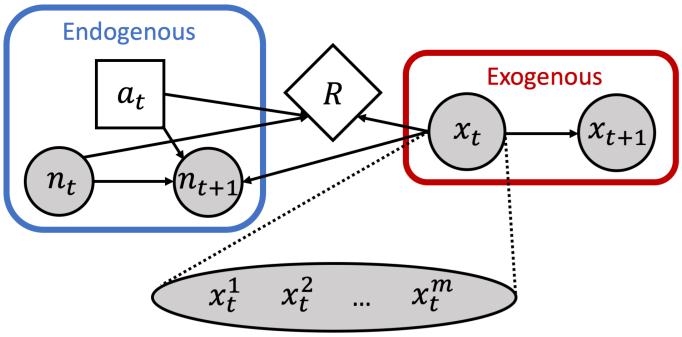
In a *factored* MDP [62], each state variable S is factored into n variables $\{S^1, \ldots, S^n\}$, where S^i has domain $\mathcal{S}^i$. A state $s \in \mathcal{S}$ is then an assignment $s = [s^1, \ldots, s^n]$ with $s^i \in \mathcal{S}^i$; thus, $\mathcal{S} \subseteq \mathcal{S}^1 \times \ldots \times \mathcal{S}^n$. Actions are similarly factored into m variables $\{A^1, \ldots, A^m\}$ with domains $\mathcal{A}^i$ so that $a \in \mathcal{A}$ is an assignment $[a^1, \ldots, a^m]$ with $a^i \in \mathcal{A}^i$; thus, $\mathcal{A} \subseteq \mathcal{A}^1 \times \ldots \times \mathcal{A}^m$. Let $V = \{S^1, \ldots, S^n\} \cup \{A^1, \ldots, A^m\}$ denote all state and action variables together. The reward function for a factored MDP is defined in terms of a subset of variables $V_{\text{rew}} \subseteq \{S^1, \ldots, S^n, A^1, \ldots, A^m\}$, which we call the *reward variables*. Variable domains may be discrete or continuous for both states and actions.

2.1.1 Context-Specific Independence in Factored MDPs

In a factored MDP, a variable is relevant for a particular task if there is *any* possibility that its value at some timestep will have an eventual influence on the value of the reward. Unfortunately, as identified by Baum et al. [16], relevance is often too conservative of a property to be useful in practice — most variables typically have *some* way of influencing the reward, under *some* sequence of actions taken by the agent.

For greater flexibility, following the work of Boutilier et al. [25], we define a *context* as a pair $(C, \mathcal{C})$, where $C \subseteq V$ is some subset of state and action variables, and $\mathcal{C}$ is a space of possible joint assignments. A state-action pair (s, a) is *in the context* $(C, \mathcal{C})$ when its joint assignment of variables in C is present in $\mathcal{C}$. Two variables $X, Y \subseteq V \setminus C$ are *contextually independent* under $(C, \mathcal{C})$ if $P(X \mid Y, C = c) = P(X \mid C = c) \; \forall \, c \in \mathcal{C}$, in which case we write $X \perp\!\!\!\perp Y \mid (C, \mathcal{C})$. This relation is called a *context-specific independence* (CSI). In Chapter 4, we will explore how CSIs can be automatically identified and exploited for planning in factored MDPs.

2.1.2 Factored MDPs with Exogenous Variables



As demonstrated in the figure on the right, some MDPs may decompose into an *endogenous* component and a (much larger) *exogenous* component. Concretely, an exogenous state variable is a state variable in a factored MDP whose dynamics are not affected by the actions of the agent. Examples include the weather and time of day. However, though they are exogenous, these processes will often play a major role in the decisions the robot should make, because they will affect the rewards (e.g., the robot should finish making dinner before evening).

Formally, we can write the state of a factored MDP at timestep t as $s_t = \begin{bmatrix} n_t & x_t \end{bmatrix}$, where n_t is the endogenous component of the state, whose transitions the agent can affect through its actions, and x_t is the exogenous component of the state. This assumption means that $P(x_{t+1} \mid x_t, a_t) = P(x_{t+1} \mid x_t)$, and therefore $P(s_{t+1} \mid s_t, a_t) = P(n_{t+1} \mid n_t, a_t, x_t) \cdot P(x_{t+1} \mid x_t)$. Because the MDP is factored, the exogenous state x_t is made up of m (not necessarily independent) state variables $x_t^1, x_t^2, \ldots, x_t^m$.

Though unaffected by the agent's actions, the exogenous state variables influence the agent through rewards and endogenous state transitions. Therefore, to plan, the

agent will need to reason about future values of the x_t. In Chapter 5, we will discuss algorithms for learning to plan efficiently in this setting.

2.2 Relational Planning Problems

While MDPs are a popular way to formalize decision-making problems, in their most general form there is very little structure to exploit, and so solving MDPs can quickly become intractable. Another formalization of decision-making that affords more structure is relational planning, where states are represented via first-order logical predicates. This structure allows for fast general-purpose solvers that can scale to solve problems involving hundreds or even thousands of actions [66].

We now formally define the notion of a relational, object-centric planning problem.

A *property* is a real-valued function on a tuple of *objects*. For example, in the expression `pose(cup3) = 5.7`, the property is `pose` and the tuple of objects is $\langle$`cup3`$\rangle$. Predicates, e.g., `on` in the expression `on(cup3, table) = True`, are a special case of properties where the output is binary.

A *relational planning problem* is a tuple $\Pi = \langle \mathcal{P}, \mathcal{A}, T, \mathcal{O}, I, G \rangle$, where $\mathcal{P}$ is a finite set of properties, $\mathcal{A}$ is a finite set of object-parameterized actions, T is a (possibly stochastic) transition model, $\mathcal{O}$ is a finite set of objects, I is the initial state, and G is the goal. A state is an assignment of values to all possible applications of properties in $\mathcal{P}$ with objects in $\mathcal{O}$. A goal is an assignment of values to any subset of the ground properties, which implicitly represents a set of states. We use $\mathcal{S}$ to denote the set of possible states and $\mathcal{G}$ to denote the set of possible goals over $\mathcal{P}$. A ground action results from applying an object-parameterized action in $\mathcal{A}$ to a tuple of objects in $\mathcal{O}$; for example, `pick(?x)` is an object-parameterized action and `pick(cup3)` is a ground action. The transition model T defines the dynamics of the environment; it maps a state, ground action, and next state to a probability.

A solution to Π can either be (1) a *plan* (a sequence of ground actions) if T is deterministic; or (2) a *policy* (a mapping from states to ground actions) if T is stochastic. A plan is a solution to Π if following the actions from the initial state

reaches a goal state. A policy is a solution to Π if executing the policy from the initial state reaches a goal state within some time horizon, with probability above some threshold; in practice, this can be approximated by sampling trajectories. In Chapter 6, we will study how to efficiently solve relational planning problems where $\mathcal{O}$ is large and contains many extraneous objects.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Inventing Symbolic, Relational Abstractions for Bilevel Planning

3.1 Motivation

We have discussed in Chapter 1 that an autonomous agent should make *good* decisions *quickly*. These two considerations are especially important when state and action spaces are continuous, and task objectives are specified with goals alone; planning can be intractable without environment-specific biases. Abstractions offer a mechanism to overcome this intractability, allowing the agent to reason at a higher level about the most salient aspects of a task [42, 44, 98, 91, 1].

State and action abstractions have a rich history in AI and robotics [115], but a major limitation of early work is the assumption that abstractions are "all you need": that planning can be decomposed into first searching for an abstract plan, and then *refining* it into an actual plan for solving the task. This *downward refinability* assumption [106] 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 (cf. task and motion planning [52, 139]). Importantly, abstractions can vary widely in how much they help (Section 3.5.2) or hinder (Section 3.6.4) bilevel planning in an environment.

Another clear limitation of early work on abstractions is the reliance on manual specification [51, 115, 106]. Even with domain expertise, this is nontrivial: it requires understanding not only the environment, but also the interplay between the abstractions and the planner. A large body of work has emerged that studies *learning* abstractions for planning [119, 74, 92], but with a few notable exceptions [13, 101, 39], these efforts assume that the learned abstractions should be "all you need."

Building on the above points, 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 chapter, we develop a framework for learning abstractions for planning that addresses all three desiderata. Specifically, *we learn state and action abstractions 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 [52]. The agent knows the transition model of the environment, but it is highly intractable to plan directly using this model in the continuous spaces, motivating the need to learn abstractions for guidance. 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 [51], and the neural components are samplers that refine the abstractions into actions that can be executed in the actual environment [29, 81, 33].

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

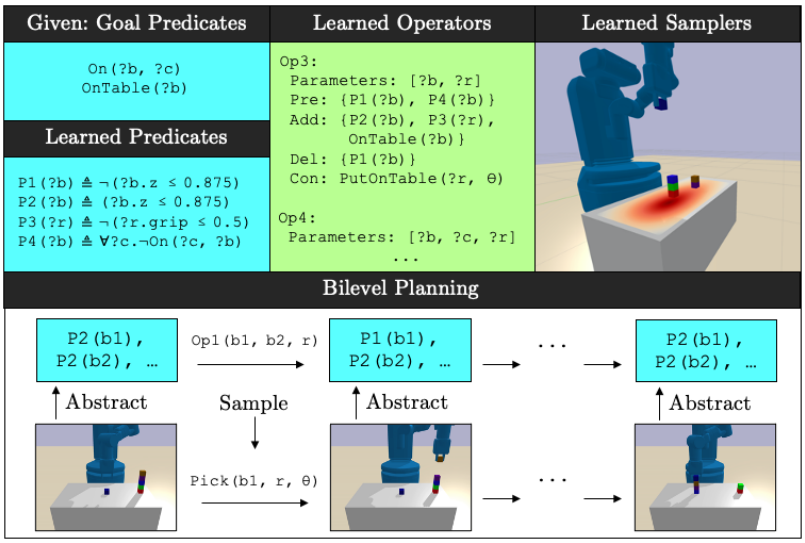


Figure 3-1: **Overview of our framework.** Given a small set of goal predicates (first panel, top), we use demonstration data to learn new predicates (first panel, bottom). In this Blocks example, implemented using the PyBullet physics simulator [37], the learned predicates `P1` – `P4` intuitively represent `Holding`, `NotHolding`, `HandEmpty`, and `NothingAbove` respectively. Collectively, the predicates define a state abstraction mapping 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.

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 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.

3.2 Related Work

Our work continues a long line of research on learning state abstractions [98, 20, 42, 44, 6, 1, 91, 78, 90, 155] and action abstractions [143, 148, 8] for planning. Most relevant are works that learn symbolic state and action abstractions compatible with AI planners [95, 74, 146, 3, 10, 9, 22, 11, 147]. Our work is particularly influenced by Pasula et al. [119], who use search through a concept language to invent symbolic state and action abstractions, and Konidaris et al. [92], who discover symbolic abstractions by leveraging the initiation and termination sets of a provided set of options [143] that satisfy an abstract subgoal property [5, 72]. However, these prior works assume the abstractions learned are "all you need" (downward refinability), and thus only perform sequential reasoning at the abstract level. Relatedly, the objectives used in these prior works are based on variations of auto-encoding, prediction error, or bisimulation, which stem from the perspective that the abstractions should *replace* planning in the original transition space, rather than *guide* it.

Recent works have also considered learning abstractions for multi-level planning, like those in the task and motion planning (TAMP) [59, 52] and hierarchical planning [19] literature. Some of these efforts consider learning symbolic action abstractions [158, 114, 137] or refinement strategies [29, 103, 30, 149, 33, 117]; our operator and sampler learning methods take inspiration from these prior works. Recent efforts by Loula et al. [101, 100] and Curtis et al. [39] consider learning both state and action abstractions for TAMP, like we do. The main distinguishing feature of our work is that our abstraction learning framework explicitly optimizes an objective that considers downstream planning efficiency.

More broadly, our work is situated within the larger research agenda of *learning to plan*. Other efforts in this space include heuristic learning [7, 134] (sometimes for TAMP [133, 82, 121]), learning to estimate the feasibility of abstract plans [46, 116], and learning to generate reductions of planning models [31, 136]. Our efforts are also related to neuro-symbolic approaches that make use of symbolic planners, often in the context of hierarchical reinforcement learning [61, 152, 88]; we differ in that the agent

knows the environment transition model in our setting, but is trying to learn effective abstractions that making planning under that model *efficient*. Finally, the related fields of inverse planning [14, 126, 157] and inverse reinforcement learning [113, 120, 94, 73] study the problem of leveraging demonstrations to infer cost functions and, in turn, policies. In our setting, we also learn from demonstration, but differ in that we assume goals are known; the unknowns are effective state and action abstractions for planning in held-out tasks.

3.3 Problem Setting

We consider learning from demonstration in deterministic planning problems. These problems are goal-based and object-centric, with continuous states and hybrid discrete-continuous actions, which makes them importantly different from relational planning problems (Section 2.2). 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 $\langle \mathcal{O}, x_0, g \rangle$. All components of environments and tasks are given to the agent.

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 dis-

crete 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.

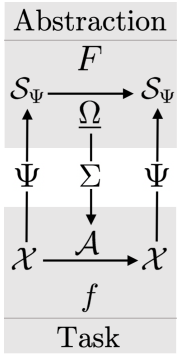
A *predicate* ψ is characterized by an ordered list of types $(\lambda_1, \ldots, \lambda_m)$ and a lifted binary state classifier $c_\psi : \mathcal{X} \times \mathcal{O}^m \to \{\text{true}, \text{false}\}$, where $c_\psi(x, (o_1, \ldots, o_m))$ is defined only when each object o_i has type λ_i. For 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 3.6.3.1. 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 plan, 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.

3.4 Relational State and Action Abstractions for Planning: Predicates, Operators, and Samplers

Since the agent has access to the transition model f, one approach for optimizing the objective described in Section 3.3 is to forgo learning entirely, and solve any held-out task by running a planner over the state state $\mathcal{X}$ and action space $\mathcal{A}$. However, searching for a solution directly in these large spaces is highly infeasible, making this approach unattractive. Instead, we propose to *learn abstractions* using the provided demonstrations. We adopt a very general definition of an abstraction [91]: mappings from $\mathcal{X}$ and $\mathcal{A}$ to alternative state and action spaces. In this section, we will describe representations that allow for fast general-purpose planning (Section 3.5), and then we will learn object-centric, relational abstractions that are optimized for efficiency when using our planning strategy (Section 3.6).



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 (as defined in Section 3.3) 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)**.** *The* abstract state s *corresponding to state* $x \in \mathcal{X}$ *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 3.6, we will discuss how the agent can use data to *invent predicates* that will make up the majority of Ψ. See Figure 3-1 (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 *respectively, 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 3-1 (second panel) for an example. Unlike in STRIPS [51], 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*:* $\underline{\text{PRE}} \subseteq s$*. If defined,* $F(s, \underline{\omega}) \triangleq (s - \underline{\text{EFF}^-}) \cup \underline{\text{EFF}^+}$*.*

(3) A mechanism for refining abstract actions into task actions. Observe that a ground operator $\underline{\omega}$ induces a partially specified controller, $C((o_1, \ldots o_v), \Theta)$ with $(o_1, \ldots o_v) = \underline{\text{PAR}_{\text{CON}}}$, 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 $\sigma : \mathcal{X} \times \mathcal{O}^{|\text{PAR}|} \to \Delta(\Theta)$*, where* $\Delta(\Theta)$ *is the space of distributions over* Θ*, the continuous parameters of the operator's controller.*

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

We denote the set of samplers as Σ. See Figure 3-1 (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 [106, 119, 74, 92], 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 $s = \text{ABSTRACT}(x, \Psi)$, $a = C((o_1, \ldots, o_v), \theta)$, and $s' = \text{ABSTRACT}(f(x, a), \Psi)$. Note that downward refinability [106] makes a much stronger assumption: that this statement holds for *every* $x \in \mathcal{X}$ where $F(s, \underline{\omega})$ is defined.

3.5 Bilevel Planning with Relational Abstractions

How can we use the components of an abstraction — predicates Ψ, operators Ω, and samplers Σ — for efficient planning? We begin by defining the notion of an *abstract plan*.

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*.*

Crucially, 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. In order to produce effective plans, we will need to appeal to hierarchical planning strategies that can reason about both abstract plans and task-level plans, and flexibly move between reasoning at either level. Therefore, we use a *bilevel planning* strategy that conducts an outer search over abstract plans using the predicates and operators, and an inner backtracking search over refinements of an abstract plan into a task solution π using the predicates and samplers.

Our bilevel planning strategy builds on the basic structure of a popular task and motion planning system devised by Srivastava et al. [139], but without the domain-specific error propagation mechanisms. Variations on this algorithm are possible, such as using local search rather than backtracking in the inner search [29], or using other optimization techniques [145, 52]. The relative performance of these algorithms will depend on the properties of the environment and tasks; we have chosen to follow the basic outline of Srivastava et al. [139] due to its relative simplicity of explanation and implementation, and due to its strong empirical performance in our experimental domains. Our initial experimentation with PDDLStream [53] (both incremental and focused) found it to be much less efficient on evaluation tasks than the planning algorithm we describe in this section.

We next describe the planning algorithm in detail.

3.5.1 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 [51] to automatically derive a domain-independent heuristic popularized by the AI planning community, such as LMCut [67]. 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 [29], we follow Srivastava et al. [139] 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 $s_i = \textsc{Abstract}(x_i, \Psi)$. 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-

[1]This usage of A* search as a generator is related to the field of top-k planning [128, 79, 127]. 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 3.6.3).

then-sample" category [139, 40, 52]. As we have described it, this planner is *not* probabilistically complete, because abstract plans are not revisited. Furthermore, there is no propagation of information about failures in the inner search back to the outer search. Extensions to ensure completeness and improve practical performance are straightforward [29], but are not our focus in this work.

3.5.2 Discussion: The Virtues of Abstractions in Bilevel Planning

We have seen in Section 3.5.1 that our bilevel planning algorithm uses search in the abstract transition space as a form of guidance. But how exactly are the state and action abstractions being leveraged? We identify four key ways:

- **Pruning unlikely candidate plans.** The REFINE procedure is only run on abstract plans $\hat{\pi}$, which by definition achieve the goal at the abstract level. Therefore, sequences of abstract actions that do not achieve the goal are never sent to REFINE. This can greatly improve efficiency because REFINE can be expensive (e.g., it may require collision checking).
- **AI planning heuristics.** Because our abstractions are symbolic (i.e., predicate- and operator-based), our A* search is informed by domain-independent AI planning heuristics. This greatly reduces the search space over the abstractions, which once again serves to reduce the number of calls to the REFINE procedure, providing large gains in efficiency.
- **Targeted samplers.** Because each ground sampler is associated with a ground operator, it specializes in producing continuous parameters that achieve that ground operator's effects. This means the ground samplers are able to guide the REFINE procedure to "follow" the expected abstract state sequence. For instance, consider an operator with two different groundings, one that has add effect `Holding(cup1)` and another that has add effect `Holding(plate1)`. When ground, the sampler for this operator can produce grasp poses specific to `cup1` and `plate1` respectively, e.g., based on object pose or shape information in the state that is

provided as input to the sampler.

- **Dense subgoal sequence.** The "expected abstract state sequence" check described in Section 3.5.1 serves as a dense subgoal sequence for the REFINE procedure, in the sense that if there is ever a deviation from it, REFINE is able to resample and/or backtrack *immediately* (rather than, say, needing to roll out trajectories up to the end of the time horizon before testing the goal [33]).

The virtues described here only hold for certain "helpful" abstractions, whereas other abstractions may be *harmful* for planning (Section 3.6.4); this is an important consideration when learning abstractions, which we discuss next.

3.6 Learning Predicates, Operators, and Samplers

Equipped with an understanding of how predicates Ψ, operators Ω, and samplers Σ can make planning efficient when they are helpful, we now describe how to *learn* all these components data-efficiently from demonstrations $\mathcal{D}$.

3.6.1 Learning Operators

We begin by describing how to learn operators Ω, assuming that the full set of predicates Ψ is already learned. In Section 3.6.3, we will describe how to invent predicates beyond the goal predicates Ψ_G.

Our operator learning method is largely based on prior work [8, 11, 33]. This method makes two restrictions on the representation that together lead to very efficient operator learning (linear time in the number of transitions in $\mathcal{D}$). First, for each CON and each possible effect set pair $(\text{EFF}^+, \text{EFF}^-)$, there is at most one operator with that $(\text{CON}, \text{EFF}^+, \text{EFF}^-)$. This restriction makes it impossible to learn multiple operators with different preconditions for the same controller and effect sets. Second, each parameter in PAR must appear in at least one of $\text{PAR}_{\text{CON}}, \text{EFF}^+$, or EFF^-. This restriction prevents modeling "indirect effects," where some object impacts the execution of a controller without its own state being changed. Though these two restrictions are limiting, we are willing to accept them because predicate invention

(Section 3.6.3) can compensate. For example, an invented predicate can quantify out an object that does not appear in the controller or the effects, to capture indirect effects.[2]

With these restrictions established, we learn operators from our demonstrations $\mathcal{D}$ and predicates Ψ in three steps. Note that each demonstration can be expressed as a sequence of transitions $\{(x, a, x')\}$, with $x, x' \in \mathcal{X}$ and $a \in \mathcal{A}$. First, we use Ψ to ABSTRACT all states x, x' in the demonstrations $\mathcal{D}$, creating a dataset of transitions $\{(s, a, s')\}$ with $s, s' \in \mathcal{S}_\Psi$. Next, we partition these transitions using the following equivalence relation: $(s_1, a_1, s_1') \equiv (s_2, a_2, s_2')$ if the effects and partially specified controllers *unify*, that is, if there exists a mapping between the objects such that a_1, $(s_1 - s_1')$, and $(s_1' - s_1)$ are equivalent to a_2, $(s_2 - s_2')$, and $(s_2' - s_2)$ respectively. This partitioning step automatically determines the number of operators that will ultimately be learned: each equivalence class will induce one operator. Furthermore, the parameters PAR, controller tuple CON, and effects $(\text{EFF}^+, \text{EFF}^-)$ of the operators can now be established as follows. For each equivalence class, we create PAR by selecting an arbitrary transition (s, a, s') and replacing each object that appears in the controller or effects with a variable of the same type (here, we are leveraging the fact that types Λ are given). This further induces a substitution $\delta : \text{PAR} \to \mathcal{O}$ for the objects $\mathcal{O}$ in this transition; the CON, EFF^+, and EFF^- are then created by applying δ to a, $(s' - s)$, and $(s - s')$ respectively. By construction, for all other transitions τ in the same equivalence class, there exists an injective substitution δ_τ under which the controller arguments and effects are equivalent to the newly created CON, EFF^+, and EFF^-. We use these substitutions for the third and final step of operator learning: precondition learning. For this, we perform an intersection over all abstract states in each equivalence class [22, 33, 39]: $\text{PRE} \leftarrow \bigcap_{\tau=(s,\cdot,\cdot)} \delta_\tau^{-1}(s)$, where $\delta_\tau^{-1}(s)$ substitutes all occurrences of the objects in s with the parameters in PAR following an inversion of δ_τ, and discards any atoms involving objects that are not in the image of δ_τ. By this construction, only the parameters in PAR will be involved in PRE, as desired.

[2]We see an example of this phenomenon in our experiments, where in the Painting environment, a quantified predicate is learned to capture whether a box lid is open or closed, as a precondition of placing something into that box.

With $\textsc{Par}, \textsc{Pre}, \textsc{Eff}^+, \textsc{Eff}^-$, and $\textsc{Con}$ now established for each equivalence class, we have completed the operators Ω.

Soundness. We note that for any predicates Ψ, the operator learning procedure is *sound* [92] over the data, in the following sense: for each transition $\tau = (x, a, x') \in \mathcal{D}$, there exists some $\underline{\omega}$, a learned operator ground with objects in x, such that $F(\textsc{Abstract}(x, \Psi), \underline{\omega})$ is defined and equals $\textsc{Abstract}(x', \Psi)$. To see this, recall that τ belongs to an equivalence class, and that this equivalence class was used to learn an operator ω. Now, we show that the desired $\underline{\omega}$ is $\langle \omega, \delta_\tau \rangle$, where δ_τ is the injective parameter-to-object substitution defined above. The $\underline{\textsc{Con}}$, $\underline{\textsc{Eff}^+}$, and $\underline{\textsc{Eff}^-}$ of $\underline{\omega}$ exactly equal those in τ, by construction of the substitution δ_τ. Additionally, because $\underline{\textsc{Pre}}$ was formed by taking an intersection of abstract states that included $\textsc{Abstract}(x, \Psi)$, it must be the case that $\underline{\textsc{Pre}} \subseteq \textsc{Abstract}(x, \Psi)$, since an intersection must be a subset of every constituent set. By Definition 4, then, the statement is satisfied. A corollary of this soundness property is that our learned abstractions are guaranteed to obey the semantics we defined in Section 3.4 with respect to the training data.

As a byproduct of operator learning, we have also determined "local" datasets for each operator, with each transition in the respective equivalence class defining an example of the operator's preconditions, controller, and effects. We will use these local datasets and the corresponding substitutions δ_τ next, as we discuss learning samplers.

3.6.1.1 Extended Example of Operator Learning Algorithm

We provide an extended example of operator learning. We give a small dataset for this example, and use it to walk through each of the three steps in the procedure.

Step 1: Generate Dataset. In this example, our demonstrations contain four transitions, which are tuples (x, a, x'). For clarity, we will not write out the task-level states x and x'. Additionally, for the sake of the example, we will assume that in this environment there is only one controller `C`, with no discrete arguments. We abstract these states with the predicate set $\Psi = \{$`Held`, `On`, `IsPurple`, `IsRed`, `IsGreen`,

IsStowable, IsStowed}, which produces four (s, a, s') tuples:

1. ({On(o_1, o_2), On(o_2, o_3), IsPurple(o_1)}, C(θ_1),{Held(o_1), On(o_2, o_3), IsPurple(o_1)})
2. ({On(o_4, o_5), On(o_5, o_6), IsRed(o_4)}, C(θ_2),{Held(o_4), On(o_5, o_6), IsRed(o_4)})
3. ({Held(o_1), IsStowable(o_1), IsGreen(o_2)}, C(θ_3),{IsStowed(o_1), IsStowable(o_1), IsGreen(o_2)})
4. ({Held(o_8), IsStowable(o_8), IsGreen(o_9)}, C(θ_4),{IsStowed(o_8), IsStowable(o_8), IsGreen(o_9)})

Intuitively, the first and second transitions might occur when picking up an object (o_1 or o_4 respectively), while the third and fourth might occur when stowing an object (o_1 or o_8 respectively). We begin by noting that we can ignore the continuous parameters θ_i of C, since they do not matter for operator learning (they would be used in sampler learning).

Step 2: Produce Equivalence Classes. Recall that two transitions are in the same equivalence class if there exists a mapping between objects such that the controller, controller discrete arguments, and effects are equivalent. Since we only have one controller C with no discrete arguments in this example, we must only check for effect equivalence. The first transition has effects $(\text{Eff}^+, \text{Eff}^-) = (\{\text{Held}(o_1)\}, \{\text{On}(o_1, o_2)\})$, while the second has effects $(\text{Eff}^+, \text{Eff}^-) = (\{\text{Held}(o_4)\}, \{\text{On}(o_4, o_5)\})$. These can be unified with the mapping $\{o_1 \leftrightarrow o_4, o_2 \leftrightarrow o_5\}$. Similarly, the third transition has effects $(\text{Eff}^+, \text{Eff}^-) = (\{\text{IsStowed}(o_1)\}, \{\text{Held}(o_1)\})$, while the fourth has effects $(\text{Eff}^+, \text{Eff}^-) = (\{\text{IsStowed}(o_8)\}, \{\text{Held}(o_1)\})$. These can be unified with the mapping $\{o_1 \leftrightarrow o_8\}$.

Note that in this unification procedure, the atoms which were unchanged, such as IsPurple(o_1), do not play a role. Furthermore, the fact that the objects are the same between transitions 1 and 3 is unimportant, because these transitions belong to different equivalence classes.

Selecting an arbitrary transition from each equivalence class and substituting objects with variables, we get the following:

- Equivalence class 1:
  - PAR: $\langle$ `?x`, `?y` $\rangle$
  - EFF$^+$: {`Held(?x)`}
  - EFF$^-$: {`On(?x, ?y)`}
  - CON: $\langle$ `C`, [] $\rangle$
  - Transitions contained: 1 and 2
  - δ_1 (substitution for transition 1): {`?x` $\rightarrow o_1$, `?y` $\rightarrow o_2$}
  - δ_2 (substitution for transition 2): {`?x` $\rightarrow o_4$, `?y` $\rightarrow o_5$}
- Equivalence class 2:
  - PAR: $\langle$ `?z` $\rangle$
  - EFF$^+$: {`IsStowed(?z)`}
  - EFF$^-$: {`Held(?z)`}
  - CON: $\langle$ `C`, [] $\rangle$
  - Transitions contained: 3 and 4
  - δ_3 (substitution for transition 3): {`?z` $\rightarrow o_1$}
  - δ_4 (substitution for transition 4): {`?z` $\rightarrow o_8$}

Note that the parameter list PAR for each equivalence class contains all parameters that appear in PAR_{CON}, EFF$^+$, or EFF$^-$.

Step 3: Learn Operator Preconditions. We now have all the ingredients of the operators except for their preconditions. For each transition in each equivalence class, we first discard any atom from the abstract state s which involves objects *not* in the image of that transition's substitution δ. For instance, the first transition has $\delta_1 = \{$`?x` $\rightarrow o_1$, `?y` $\rightarrow o_2\}$. The image is $\{o_1, o_2\}$, which excludes o_3. This means that the atom `On`(o_2, o_3) is discarded from s.

After discarding atoms appropriately, we end up with these abstract states for each transition:

1. {`On`(o_1, o_2), `IsPurple`(o_1)}
2. {`On`(o_4, o_5), `IsRed`(o_4)}
3. {`Held`(o_1), `IsStowable`(o_1)}
4. {`Held`(o_8), `IsStowable`(o_8)}

Now, the preconditions for each equivalence class are obtained by applying each δ_i^{-1} to these abstract states and taking intersections [22, 33, 39]. This produces the final operator set Ω, which does not contain any extraneous atoms related to object color:

- Operator 1 (from equivalence class 1):
  - PAR: ⟨ `?x`, `?y` ⟩
  - PRE: {`On(?x, ?y)`}
  - EFF$^+$: {`Held(?x)`}
  - EFF$^-$: {`On(?x, ?y)`}
  - CON: ⟨ `C`, [] ⟩
- Operator 2 (from equivalence class 2):
  - PAR: ⟨ `?z` ⟩
  - PRE: {`Held(?z)`, `IsStowable(?z)`}
  - EFF$^+$: {`IsStowed(?z)`}
  - EFF$^-$: {`Held(?z)`}
  - CON: ⟨ `C`, [] ⟩

3.6.2 Learning Samplers

The role of a sampler $\sigma \in \Sigma$ is to *refine* its associated operator ω, suggesting continuous parameters of actions that will transition the environment from a state

where the preconditions hold to a state where the effects follow. Recall that a sampler $\sigma : \mathcal{X} \times \mathcal{O}^{|\textsc{Par}|} \to \Delta(\Theta)$ defines a conditional distribution $P(\theta \mid x, o_1, \ldots, o_k)$, where θ are continuous parameters for the controller C in ω, and $(o_1, \ldots, o_k)$ represent a set of objects that could be used to ground ω, with $|\textsc{Par}| = k$. We learn samplers of the following form, one per operator:

$$\sigma(x, o_1, \ldots, o_k) = r_\sigma(x[o_1] \oplus \cdots \oplus x[o_k]),$$

where $x[o]$ denotes the feature vector for o in x, the $\oplus$ denotes concatenation, and r_σ is the model to be learned.

From the local datasets created during operator learning (Section 3.6.1), we can create datasets for supervised sampler learning, with one dataset per sampler. Consider any (non-abstract) transition $\tau = (x, a, \cdot)$ in the equivalence class associated with an operator ω. To create a data point for the associated sampler, we can reuse the substitution δ_τ found during operator learning to create an input vector $x[\delta_\tau(v_1)] \oplus \cdots \oplus x[\delta_\tau(v_k)]$, where $(v_1, \ldots, v_k) = \textsc{Par}$. The corresponding output for supervised learning is the continuous parameter vector θ in the action a.

With these datasets created, one could use any method for multidimensional distributional regression to learn each r_σ. In this work, we learn two neural networks to parameterize each sampler. The first neural network takes in $x[o_1] \oplus \cdots \oplus x[o_k]$ and regresses to the mean and covariance matrix of a Gaussian distribution over θ; here, we are assuming that the desired distribution has nonzero measure, but the covariances can be arbitrarily small in practice. This neural network is a sampler in its own right, but its expressive power is limited, e.g., to unimodal distributions. To improve representational capacity, we learn a second neural network that takes in $x[o_1] \oplus \cdots \oplus x[o_k]$ *and* θ, and returns true or false. This binary classifier is then used to rejection sample from the Gaussian distribution produced by the first neural network. To create negative examples for the classifier for any operator, we use all transitions τ' such that the controller in τ' matches that in $\textsc{Con}$, but the effects in τ' are different from $(\textsc{Eff}^+, \textsc{Eff}^-)$. These two neural networks resemble the generator

and discriminator used in generative adversarial networks [81, 58], but we train them separately for simplicity and stability.

```
PLAN(x_0, g, Ψ, Ω, Σ)
    // Parameters: n_abstract, n_samples.
1   s_0 ← ABSTRACT(x_0, Ψ)
2   for π̂ in GENABSTRACTPLAN(s_0, g, Ω, n_abstract) do
3       if REFINE(π̂, x_0, Ψ, Σ, n_samples) succeeds w/ π then
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. See Section 3.5.1 for details. The use of an abstraction helps guide planning in the continuous task spaces, making it more efficient in several ways, as discussed in Section 3.5.2.

```
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 3.6.3.1 for details.

3.6.3 Inventing Predicates via Local Search over a Grammar

We have described how we can learn action abstractions and samplers for refinement when a state abstraction Ψ is provided. Now, we describe how such a state abstraction can be learned via predicate invention, completing our pipeline.

Inspired by prior work [22, 101, 39], we approach the predicate invention problem from a program synthesis perspective [140, 97, 38, 49]. 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, then work backward from there, in Sections 3.6.3.1 through 3.6.3.3.

3.6.3.1 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:

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

where Ω and Σ are learned using Ψ as we described in Sections 3.6.1 and 3.6.2, PLAN is the algorithm described in Section 3.5, and TIME$(\cdot)$ measures the time that PLAN takes to find a solution.[3] 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 the ordering over predicate sets, 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 [119, 137], bisimulation [22, 39], and inverse planning-based objectives [120, 157], but found them all to be divergent from J_{real}, leading to poor performance (see the baselines

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

in Section 3.7). 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:

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

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. See Algorithm 2 for pseudocode of the key method, EstimateTotalPlanningTime.

To estimate the total planning time, we perform the same A* abstract search described in Section 3.5.1, 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 3.5.1). We use a geometric distribution in particular to allow for some difference in cost while making the probability drop off steeply – this provides robustness to small degrees of suboptimality in the demonstrations. If costs are not integers, one could use an exponential distribution instead. 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. While more sophisticated strategies could be helpful in general, this simple and cheap procedure worked well in our experiments.

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 of this quantity, to avoid noise due to CPU speed. 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 based on the predicate set.

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). For instance, *no* abstract plan may be found under the current predicate set. 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 this over all the training tasks.

What is the ideal choice for n_{abstract}, the maximum number of abstract plans to consider within ESTIMATETOTALPLANNINGTIME? From an efficiency perspective, $n_{\text{abstract}} = 1$ is ideal, but otherwise, it is not obvious whether to prefer the value of n_{abstract} that will eventually be used with PLAN at evaluation time, or to instead prefer $n_{\text{abstract}} = \infty$. On one hand, we want ESTIMATETOTALPLANNINGTIME to be as much of a mirror image of PLAN as possible; on the other hand, some experimentation we conducted suggests that a larger value of n_{abstract} can smooth the objective landscape, which makes search easier. In practice, it may be advisable to treat n_{abstract} as a hyperparameter of the learning algorithm.

The proxy objective J_{proxy} we proposed calculates and combines two characteristics of a candidate predicate set Ψ: (1) abstract plan cost "error," i.e., $|\textsc{Cost}(\hat{\pi}) - \textsc{Cost}(\pi^*)|$; and (2) abstract planning time, i.e., number of nodes created during A*. The first feature uses only the costs of the demonstrated plans, while the second feature does not use the demonstrated plans at all. In Figure 3-4, we conduct an empirical analysis to further unpack the contribution of these two features to the overall proxy objective, finding them to be helpful together but insufficient individually.

3.6.3.2 Local Search over Candidate Predicate Sets using the Proxy Objective

With our proxy objective J_{proxy} established, we turn to the question of how to best optimize it. The empirical analysis in Figure 3-4 shows that J_{proxy} monotonically improves as predicates are added one at a time to Ψ, which motivates the idea of using local search as our optimizer. Specifically, 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 3-2 for a real example of predicate invention via hill climbing search, taken from our experiments in the Blocks environment.

3.6.3.3 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 with the following production rules, similar to those of Pasula et al. [119], 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, ..., 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, starting

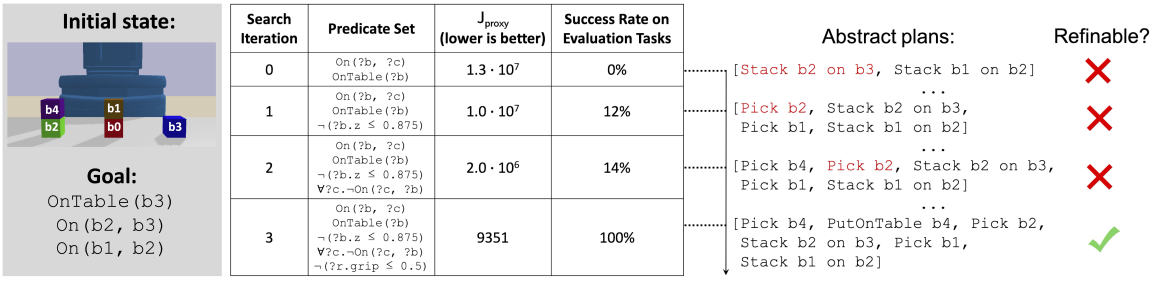


Figure 3-2: **Predicate invention via hill climbing**. (Left) An example task in Blocks. (Middle) Hill climbing over predicate sets, starting with the goal predicates Ψ_G. On each iteration, the single predicate that improves J_{proxy} the most is added to the set. The rightmost table column shows success rates under a 10-second timeout on a held-out set of evaluation tasks. These tasks are *not* part of the learning algorithm; we include them here only to make the point that the evaluation performance improves as J_{proxy} decreases. (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 (shown in red) is, thus, 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` in the initial state. At iteration 2, a predicate with the intuitive meaning `NothingAbove` is added, which allows the agent to realize that it must move `b4` out of the way if it wants to pick 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. This example shows that the predicates "prune away" unrefinable abstract plans, making planning more effective and efficient.

from the median values in $\mathcal{D}$. As an example, a type `block` might have a feature dimension corresponding to its `size`, and a classifier could be `block.size` $\leq$ `0.5`. All goal predicates have cost 0. All single-feature inequality classifiers have cost computed based on the normalized constant, with cost 0 for constant 0.5, cost 1 for constants 0.25 and 0.75, cost 2 for constants 0.125, 0.375, 0.625, 0.875, etc. This provides a bias toward predicates that split on the median value seen in the data.

- We include all negations of predicates in the base grammar. Negating a predicate adds a cost of 1.
- We include two types of universally quantified predicates over the predicates thus

far: (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)`. Universally quantifying adds a cost of 1.

- We include all negations of universally quantified predicates. Negating a predicate adds a cost of 1.
- Following prior work [39], 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 $\mathcal{D}$. Finally, we discard the goal predicates Ψ_G from the grammar, since they are included in every candidate predicate set Ψ of our search already.

Note that there are many concepts this grammar cannot represent, such as relationships between two objects or two features of the same object, but it is nevertheless rich enough to capture a wide class of state abstractions in practice. For instance, existential quantification can be expressed by composing a negation, a universal quantification, and another negation.

The costs accumulated over the production rules lead us to a final cost associated with each predicate ψ, denoted $\text{PEN}(\psi)$, where a higher cost represents a predicate with higher complexity. These costs are akin to (unnormalized) negative log probabilities in a probabilistic context-free grammar (PCFG). For improved generalization to held-out tasks, we use the costs to regularize J_{proxy} during local search, with a weight small enough to primarily prevent the addition of "neutral" predicates that neither harm nor hurt J_{proxy}. The regularization term is:

$$J_{\text{reg}}(\Psi) \triangleq w_{\text{reg}} \sum_{\psi \in \Psi} \text{PEN}(\psi),$$

where $w_{\text{reg}} = 10^{-4}$ in our experiments. To generate our candidate predicate set for local search (Section 3.6.3.2), we enumerate n_{grammar} predicates from the grammar, in order of increasing cost. In our experiments, we use $n_{\text{grammar}} = 200$.

3.6.4 Discussion: The Potential Vices of Learned Abstractions in Bilevel Planning

In Section 3.5.2, we saw several ways that abstractions can aid bilevel planning. As we conclude our discussion on learning, we emphasize that not all abstractions are equally helpful; in fact, some can be actively harmful, for several reasons:

- **Abstract state space can become large.** With each new predicate added to Ψ during predicate invention, the size of the abstract state space grows combinatorially, slowing down abstract search.
- **Abstract action space can become large.** Similarly, additional predicates added to Ψ will often lead to additional operators learned, which grows the abstract action space, in turn increasing the branching factor of abstract search.
- **Bilevel planning can spend more time on worse abstract plans.** Even with constant size, different state and action abstractions can lead to radically different planning performance. For example, in the abstract search, the heuristic induced by the learned predicates and operators can have an enormous impact on the time required to find abstract plans; also, those abstract plans will vary in refinability depending on the predicates and operators.
- **Generalization may suffer.** If abstraction learning overfits to the training data, planning may fail to generalize to held-out evaluation tasks. For example, overly specific preconditions may prevent the use of a necessary controller.

Furthermore, the vices or virtues of any given abstraction depend on the properties of the environment, task distribution, and planner. These observations make a strong case for the type of abstraction learning we have discussed in this section, which is both "task-aware" and "planner-aware," reasoning explicitly about optimizing the efficiency of the planner.

3.7 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?

3.7.1 Experimental Design

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. We now describe the environments. The first three environments were established in prior work by Silver et al. [137], but in that prior 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. The three object types are block, target, and robot. Blocks and targets have two features for their pose and width, and robots have one feature for the gripper joint state. The block widths are larger than the target widths, and the goal requires each block to be placed so that it completely covers the respective

target region, so $\Psi_G = \{\texttt{Covers}\}$, where `Covers` is an arity-2 predicate. There is only one controller, `PickPlace`, with no discrete arguments; its Θ is a single real number denoting the location to perform either a pick or a place, depending on the current state of the robot's gripper. Each action updates the state of at most one block – the block that is closest to the location θ, so long as it is within a threshold of that location (if no block is within this threshold, the state is unchanged). Both training tasks and evaluation tasks involve 2 blocks, 2 targets, and 1 robot. In each task, with 75% probability the robot starts out holding a random block; otherwise, both blocks start out on the table. 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. The two object types are block and robot. Blocks have four features: an x/y/z pose and a bit for whether it is currently grasped. Robots have four features: x/y/z end effector pose and the (symmetric) value of the finger joints. The goals involve assembling towers, so $\Psi_G = \{\texttt{On}, \texttt{OnTable}\}$, where the former has arity 2 and describes one block being on top of another, while the latter has arity 1. There are three controllers: `Pick`, `Stack`, and `PutOnTable`. `Pick` is parameterized by a robot and a block to pick up. `Stack` is parameterized by a robot and a block to stack the currently held one onto. `PutOnTable` is parameterized by a robot and a 2D place pose representing normalized coordinates on the table surface at which to place the currently held block. Training tasks involve 3 or 4 blocks, while evaluation tasks involve 5 or 6 blocks; all tasks have 1 robot. In all tasks, all blocks start off in collision-free poses on the table. 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. The five object types are widget, box, shelf, box lid, and robot. Widgets have eight features: an x/y/z pose, a dirtiness level (requiring washing), a wetness level (requiring drying), a color, a bit for whether it is currently grasped, and the 1D

gripper rotation with which it is grasped if so. Boxes and shelves have one feature for their color. Box lids have one feature for whether or not they are open. Robots have one feature for the gripper joint state. The goals involve painting the widgets to be the same color as either a box or a shelf, and then placing each widget into the appropriate one, so $\Psi_G = \{\texttt{InBox}, \texttt{InShelf}, \texttt{IsBoxColor}, \texttt{IsShelfColor}\}$, all of which have arity 2 (a widget, and either a box or a shelf). There are two physical constraints in this environment: (1) placing into a box can only succeed if the robot is top-grasping a widget, while placing into a shelf can only succeed if the robot is side-grasping it; (2) a box can only be placed into if its respective lid is open. There are six controllers: `Pick`, `Wash`, `Dry`, `Paint`, `Place`, and `OpenLid`. All six are discretely parameterized by a robot argument; `Pick` is additionally parameterized by a widget to pick up, and `OpenLid` by a lid to open. `Pick` has 4 continuous parameters: a 3D grasp pose delta from that widget's center of mass, and a gripper rotation. `Wash`, `Dry`, and `Paint` have 1 continuous parameter each: the amount of washing, the amount of drying, and the desired new color, respectively. `Place` has 3 continuous parameters: a 3D place pose corresponding to where the currently held widget should be placed. Training tasks involve 2 or 3 widgets, while evaluation tasks involve 3 or 4 blocks; all tasks have 1 box, 1 shelf, and 1 robot. In each task, with 50% probability the robot starts out holding a random widget; otherwise, all widgets start out on the table. Also, in each task, with 30% probability the box lid starts out open. 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 learner's ability to be robust to an insurmountable lack of downward refinability. The eight object types are contraption, screw, nail, bolt, screwdriver, hammer, wrench, and robot. Contraptions have two features: an x/y pose. Screws, nails, bolts, and the three tools have five features: an x/y pose, a shape, a

size, and a bit indicating whether it is held. Robots have one feature for the gripper joint state. The goals involve fastening the screws, nails, and bolts onto target contraptions, so $\Psi_G = \{\texttt{ScrewPlaced}, \texttt{NailPlaced}, \texttt{BoltPlaced}, \texttt{ScrewFastened}, \texttt{NailFastened}, \texttt{BoltFastened}\}$. The first three have arity 2 (a screw/nail/bolt and which contraption it is placed on); the last three have arity 1. There are three physical constraints in this environment: (1) a screwdriver can only be used to fasten a screw if its shape is close enough to that of the screw; (2) some screws have a shape that does not match any screwdriver's, and so these screws must be fastened by hand; (3) the three tools cannot be picked up if their sizes are too large. There are eleven controllers: `Pick{Screw, Nail, Bolt, Screwdriver, Hammer, Wrench}`, `Place`, `FastenScrewWithScrewdriver`, `FastenScrewByHand`, `FastenNailWithHammer`, and `FastenBoltWithWrench`. All eleven are discretely parameterized by a robot argument; `Pick` controllers are additionally parameterized by an object to pick up, and `Fasten` controllers by a screw/nail/bolt and tool (except `FastenScrewByHand`, which does not have a tool argument). `Place` has 2 continuous parameters: a 2D place pose corresponding to where the currently held object should be placed, which can be either onto the table or onto a contraption (only if the currently held object is not a tool). Training tasks involve 2 screws/nails/bolts and 2 contraptions, while evaluation tasks involve 2 or 3 screws/nails/bolts and 3 contraptions; all tasks have 3 screwdrivers, 2 hammers, 1 wrench, and 1 robot. Evaluation tasks require 7-20 actions to solve.

Methods. The methods we evaluate are: ours, 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* [55], as in prior work [39]. 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. Note that because goal predicates are given, the criterion

regarding goal distinguishability is satisfied under any abstraction.

- **Branching.** A baseline that learns abstractions by optimizing the *branching factor* of planning. Specifically, this baseline learns abstractions that aim to minimize the total number of applicable operators over states in the demonstrations.
- **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. This baseline is inspired by work on inverse planning [14, 157].
- **GNN Shooting.** A baseline that trains a graph neural network [15] 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 produced continuous arguments as the mean of a Gaussian with fixed variance. We use the known transition model f to check if the goal is achieved, and repeat until the planning timeout is reached.
- **GNN Model-Free.** A baseline that uses the same trained GNN as above, but at evaluation time, directly executes the policy. This has the advantage of being more efficient during evaluation than GNN Shooting, but is less effective.
- **Random.** A baseline that simply executes a random controller with random arguments on each step. This baseline does not do any learning.
- **Manual.** An oracle approach that plans with manually designed state abstractions (predicates) for each environment. Operators and samplers are still learned.
- **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.

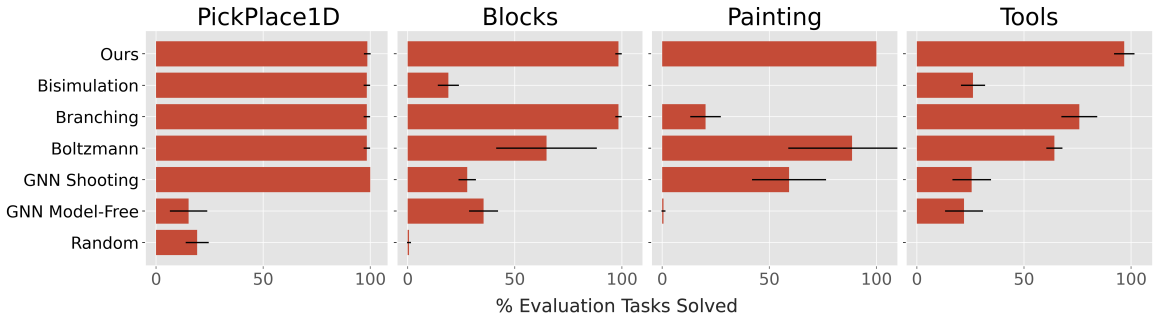


Figure 3-3: **Ours versus baselines.** Percentage of 50 evaluation tasks solved under a 10-second timeout, for all four environments. All results are averaged over 10 seeds. Horizontal black bars denote standard deviations. Learning times and additional metrics are reported in Section 3.7.3. We can see that our learned abstractions (Ours) perform extremely well compared to all six baselines.

Additional details. All sampler neural networks are fully connected, with two hidden layers of size 32 each, and trained with the Adam optimizer [84] 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 [35]. 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 [4]; 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.

3.7.2 Main Results and Discussion

We provide real examples of learned predicates and operators for all environments in Section 3.7.3.

Comparisons with baselines are shown in Figure 3-3, 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. The lackluster performance of the bisimulation baseline

	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 3.1: **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). All results are averaged over 10 seeds. The Node and Time columns average over *solved tasks only*. Standard deviations are provided in Section 3.7.3. These ablations confirm that abstraction learning and bilevel planning contribute to the strong performance of Ours.

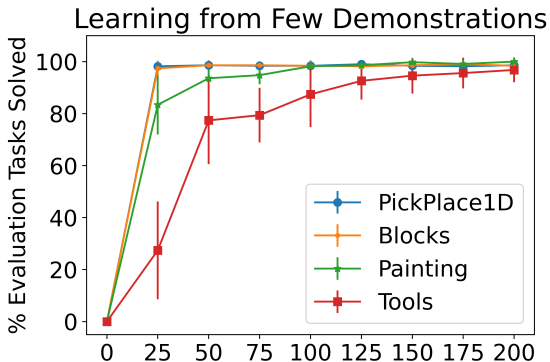
is especially notable because of its prevalence in the literature [119, 74, 22, 39]. We examined its failure modes more closely and found that it consistently selects good predicates, but not *enough* of them. This is because requiring the operators to be a perfect predictive model in the abstract spaces is often not enough to ensure good planning performance. For example, in the Blocks environment, the goal predicates together with the predicate `Holding(?block)` are enough to satisfy bisimulation on our data, while other predicates like `Clear(?block)` and `HandEmpty()` are useful from a planning perspective. Considering now the GNN baselines, we see that while shooting with the transition model f is beneficial versus using the GNN as a raw policy, the performance is generally far worse than Ours. Additional experimentation we conducted suggests that the GNN gets better with around an order of magnitude more data; this is consistent with previous findings that such strategies are very data-hungry [110].

Table 3.1 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. To understand this, let us walk through an example from PickPlace1D, where our learned abstractions lead to a 10x improvement in planning time. Our manually designed predicate set contained `Held(?block)` and `HandEmpty()`, in addition to the always-given goal predicate `Covers(?block, ?target)`. In addition to inventing two predicates that are equivalent to `Held` and `HandEmpty`, Ours in-

vented two more: `P3(?block)` $\triangleq \forall$`?t` . $\neg$`Covers(?block, ?t)`, and `P4(?target)` $\triangleq \forall$`?b` . $\neg$`Covers(?b, ?target)`. Intuitively, `P3` means "the given block is not on any target," while `P4` means "the given target is clear." These predicates turn out to be very useful from an abstract search perspective, because there is no use in moving a block once it is already on a target, and the robot cannot place a block on a target that is already occupied. So, the operators have more restrictive preconditions under the learned predicates, causing the A* search to have lower branching factor. Furthermore, the inclusion of `P4` prevents the planner from considering a non-refinable abstract plan that "parks" a held object on a target where another object must later be placed. This example illustrates that the learned abstractions are tailored to the task distribution; that is, our learning method is *task-aware*.

Next, we look at the performance of the ablations in Table 3.1. 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 [137]. Upon inspection, we find that Ours learns abstractions that *are* downward refinable. For PickPlace1D, the learned "target clear" predicate leads to downward refinability, as discussed with `P4` in the previous paragraph. For Painting, a learned "box lid open" predicate resolves the downward refinability issue discussed in prior work [137], where the position of the box lid (open or closed) was not modeled in the manual abstraction. By 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, and this performance typically improves as the number of demonstrations increases. Generally, providing more demonstrations helps the following aspects of our system: (1) inventing fewer extraneous, overfitted predicates; (2) learning more accurate operator preconditions; (3) allowing the sampler neural networks to be trained on a larger amount of data.

To address **(Q4)**, the table on the right shows an additional experiment we conducted, where we varied the AI planning heuristic used by the GenAbstractPlan routine of our bilevel planner in the Blocks environment. Recall that our predicate invention method uses GenAbstractPlan as well, so it too is affected by this heuristic change. All numbers show a mean over 10 seeds. Interestingly, while the gap in performance is limited when using LMCut, our system shows a massive improvement (over 30x fewer nodes created) versus Manual when using hAdd. These results are especially surprising because A* with hAdd is generally considered inferior to other heuristic search algorithms.[4] Inspecting the learned abstractions, we find that our approach invents four unary predicates with the intuitive meanings `Holding`, `NothingAbove`, `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) [12], we see the following differences: `Clear` is omitted[5], and `NothingAbove` and `NotOnAnyBlock` are added.

	Ours			**Manual**		
Heuristic	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

We observe that `NothingAbove` and `NotOnAnyBlock` are logical transformations of predicates used in the standard IPC representation. This motivated us to run a

[4]We also experimented with GBFS instead of A*, and hFF, hSA, and hMax instead of hAdd. A* with hAdd performed the best by far.

[5]In the standard encoding, "clear" means "nothing above and not holding."

separate, symbolic-only experiment, where we collected IPC blocks world problems and transformed them to use these learned predicates. We found that using A* and hAdd, planning with our learned representations is much faster than planning with the IPC representations. For example, in the hardest problem packaged with Pyperplan, which contains 17 blocks, planning with our operators requires approximately 30 seconds and 841 node expansions, whereas planning with the standard encoding requires 560 seconds and 17,795 expansions. We also tried Fast Downward [66] (again with A* and hAdd) on a much harder problem from IPC 2000 with 36 blocks. With our learned representations, planning succeeds in 12.5 seconds after approximately 7,000 expansions, whereas under the standard encoding the planner fails to find a plan within a 2 *hour* timeout. *See Section 3.7.4 for the complete operators, as well as further insight into this discrepancy.* Note that all of these results are specific to Blocks, A*, and hAdd, and that is exactly the point: even when using an unconventional combination of search algorithm and heuristic, our *planner-aware* method learns abstractions that optimize the efficiency of the given planner in the given environment.

3.7.3 Additional Results

Table 3.2 provides learning times for all experiments. Tables 3.3 and 3.4 report nodes created and wall-clock time respectively for all evaluation tasks.

Figure 3-4 analyzes the two main features used by our proxy objective function: plan cost error and abstract search time. It shows that neither of these components is sufficient on its own for making our predicate invention pipeline work.

We now go through each of our four environments, providing an example of learned predicates and operators from a single seed randomly chosen among successful ones. We also provide additional statistics for our main method, to supplement the data presented in the main text. Note that the evaluation plan length statistics are averaged over both 10 seeds and 50 evaluation tasks per seed, *with standard deviations over seed only.*

Environment	Ours	Bisimulation	Branching	Boltzmann	GNN Sh	GNN MF	Manual	No Invent
PickPlace1D	625 (134)	176 (3)	219 (145)	264 (17)	1951 (85)	1951 (85)	177 (147)	66 (0)
Blocks	10237 (853)	800 (44)	1561 (98)	9798 (1688)	4047 (209)	4047 (209)	102 (5)	84 (2)
Painting	18395 (28153)	872 (380)	2883 (144)	9457 (3421)	9185 (166)	9185 (166)	565 (457)	260 (2)
Tools	18666 (2815)	573 (20)	5524 (747)	9716 (1000)	7362 (197)	7362 (197)	167 (3)	141 (3)

Table 3.2: **Learning times in seconds for all experiments**. All numbers are means over 10 seeds, with standard deviations in parentheses. For the GNN-based methods, learning time encompasses training the neural networks. For the other methods, learning time encompasses learning predicates, operators, and samplers (i.e., all components of the abstraction). Even though our main method performs well (Ours), this does come at the cost of increased learning time (although the learning is purely offline). Note that the Manual approach only manually specifies a *state* abstraction (predicates); operators and samplers must still be learned, contributing to the non-zero learning time. Thus, comparing Ours and Manual shows that the large majority of learning time in our system is spent on predicate invention.

Environment	Ours	Bisimulation	Branching	Boltzmann	Manual	Down Eval	No Invent
PickPlace1D	4.8 (0.2)	4.7 (0.2)	4.7 (0.2)	5.3 (0.2)	6.5 (0.3)	4.8 (0.2)	14.1 (4.0)
Blocks	2948.5 (1293.2)	46.9 (18.0)	2948.5 (1293.2)	7844.0 (6655.4)	2940.5 (1299.1)	2948.5 (1293.2)	427.7 (83.7)
Painting	501.8 (180.0)	–	876.6 (509.7)	4008.8 (3851.3)	2607.5 (1117.2)	489.0 (190.2)	–
Tools	1897.2 (1404.0)	5247.7 (2560.6)	167.8 (78.4)	909.9 (174.1)	4770.9 (886.8)	152.5 (27.6)	–

Table 3.3: **Number of nodes created by abstract search during planning in evaluation tasks**. All numbers are means over *solved tasks only* across 10 seeds, with 50 evaluation tasks per seed, and with standard deviations in parentheses.

3.7.3.1 PickPlace1D

Statistics for our main method, averaged over 10 random seeds (standard deviations parenthesized):

- Average number of predicates in Ψ (both invented and goal predicates): 5.9 (0.54)
- Average number of operators in Ω: 2.1 (0.3)
- Average plan length during evaluation: 2.44 (0.09)

See Figure 3-5 for example learned predicates and operators for a randomly chosen successful seed.

3.7.3.2 Blocks

Statistics for our main method, averaged over 10 random seeds (standard deviations parenthesized):

- Average number of predicates in Ψ (both invented and goal predicates): 6.0 (0.0)
- Average number of operators in Ω: 4.0 (0.0)
- Average plan length during evaluation: 9.17 (0.69)

Environment	Ours	Bisimulation	Branching	Boltzmann	GNN Sh	GNN MF	Random	Manual	Down Eval	No Invent
PickPlace1D	0.006 (0.0)	0.006 (0.0)	0.006 (0.0)	0.005 (0.0)	0.436 (0.1)	0.014 (0.0)	0.004 (0.0)	0.045 (0.0)	0.008 (0.0)	1.369 (0.6)
Blocks	0.296 (0.1)	0.158 (0.1)	0.284 (0.1)	0.954 (0.3)	0.138 (0.1)	0.249 (0.1)	0.006 (0.0)	0.251 (0.1)	0.318 (0.1)	1.235 (1.3)
Painting	0.470 (0.2)	–	4.186 (0.9)	0.600 (0.3)	2.077 (1.2)	0.073 (0.0)	–	0.464 (0.1)	0.208 (0.0)	–
Tools	0.457 (0.3)	0.699 (0.3)	0.109 (0.0)	0.247 (0.0)	0.311 (0.2)	0.043 (0.0)	–	0.491 (0.1)	0.060 (0.0)	–

Table 3.4: **Total time in seconds for evaluation tasks**. These results encompass planning time (when applicable) and policy or plan inference time (the time taken to produce an action at each step, given the current state). All numbers are means over *solved tasks only* across 10 seeds, with 50 evaluation tasks per seed, and with standard deviations in parentheses.

See Figure 3-6 for example learned predicates and operators for a randomly chosen successful seed.

3.7.3.3 Painting

Statistics for our main method, averaged over 10 random seeds (standard deviations parenthesized):

- Average number of predicates in Ψ (both invented and goal predicates): 22.1 (1.45)
- Average number of operators in Ω: 11.2 (0.6)
- Average plan length during evaluation: 14.76 (0.29)

See Figures 3-7 and 3-8 for example learned predicates and operators for a randomly chosen successful seed.

3.7.3.4 Tools

Statistics for our main method, averaged over 10 random seeds (standard deviations parenthesized):

- Average number of predicates in Ψ (both invented and goal predicates): 27.4 (4.39)
- Average number of operators in Ω: 17.8 (0.98)
- Average plan length during evaluation: 10.1 (0.12)

See Figures 3-9 and 3-10 for example learned predicates and operators for a randomly chosen successful seed.

3.7.4 More Explanation of Blocks / hAdd Results

Why exactly do our learned predicates and operators outperform the standard ones when planning with A* and hAdd? First, we note that it is highly uncommon to use hAdd with A* in practice with hand-defined PDDL representations, because hAdd is inadmissible and suffers greatly from overestimation issues [21]. Nevertheless, the interesting phenomenon in our work is that our system is able to *learn an abstraction* that copes with the faults of this combination of search algorithm and heuristic. To understand this further, we make the following observations:

- In both cases, the planner must escape from a local minimum with almost every pick operation. For example, in a small problem with 5 blocks where the hand is initially empty, the hAdd values of the states in the plan found are [9, 13, 9, 11, 6, 8, 4, 5, 2, 1, 0] when planning with the standard operators, and [14, 16, 11, 10, 6, 7, 4, 4, 2, 1, 0] when planning with our learned operators. Note the alternation of increasing and decreasing values; the ideal scenario for planning would instead be that these values decrease monotonically.

- In states that follow a pick, the hAdd values consistently *overestimate* the true cost-to-go, in both cases. For example, after the first pick with the standard operators, the hAdd value and true cost-to-go are 13 and 9 respectively; for the learned operators, they are 16 and 9 respectively.

- Here is the main difference: in states that *precede* a pick, the hAdd values from the standard operators sometimes *underestimate* the true cost-to-go. In the example above, the initial state has an hAdd value of 9, but the true cost-to-go is 10. In harder problems, these underestimations occur with higher frequency; for example, in a problem with 20 blocks, there are 8 cases in the plan found where states preceding picks underestimate the true cost-to-go. By contrast, the hAdd values from our *learned* operators do not ever seem to underestimate the true cost-to-go, in the problems that we analyzed.

- Furthermore, this underestimation occurs regularly in states that are local min-

ima, immediately preceding states where the heuristic will be an overestimate, so A* struggles greatly. Since nodes are expanded in order of $f = g + h$, that is, cost of the plan so far plus heuristic value, A* will spend time exploring large subtrees rooted at nodes that underestimate true cost-to-go before moving onto the nodes that overestimate it, including those that will ultimately be included in the plan.

For reproducibility, we provide the complete operators used to conduct this experiment. We started from the standard blocks domain PDDL downloaded from the `planning.domains` Github repository, removed the `Clear` predicate, and added the two predicates our system learned, with the intuitive meanings `NothingAbove` and `NotOnAnyBlock`. Problem files were updated accordingly. We ran Fast Downward with the `-search astar(add)` option. Here is the domain file, with changes highlighted in red (deletion) and green (addition):

```
(define (domain blocks)
(:predicates
    (on ?v0 ?v1)
    (ontable ?v0)
    (clear ?v0)
    (nothingabove ?v0)
    (notonanyblock ?v0)
    (handempty)
    (holding ?v0)
)
(:action pick-up
    :parameters (?x)
    :precondition (and
        (clear ?x)
        (nothingabove ?x)
        (notonanyblock ?x)
        (ontable ?x)
        (handempty))
    :effect (and
        (not (clear ?x))
        (not (ontable ?x))
        (not (handempty))
        (holding ?x))
)
(:action put-down
    :parameters (?x)
    :precondition (and
        (holding ?x)
        (nothingabove ?x)
        (notonanyblock ?x))
    :effect (and
        (clear ?x)
        (not (holding ?x))
        (handempty)
        (ontable ?x))
)

(:action stack
    :parameters (?x ?y)
    :precondition (and
        (holding ?x)
        (clear ?y)
        (nothingabove ?x)
        (notonanyblock ?x)
        (nothingabove ?y))
    :effect (and
        (not (holding ?x))
        (not (clear ?y))
        (clear ?x)
        (not (nothingabove ?y))
        (not (notonanyblock ?x))
        (handempty)
        (on ?x ?y))
)
(:action unstack
    :parameters (?x ?y)
    :precondition (and
        (on ?x ?y)
        (clear ?x)
        (nothingabove ?x)
        (handempty))
    :effect (and
        (holding ?x)
        (clear ?y)
        (not (clear ?x))
        (nothingabove ?y)
        (notonanyblock ?x)
        (not (handempty))
        (not (on ?x ?y)))
))
```

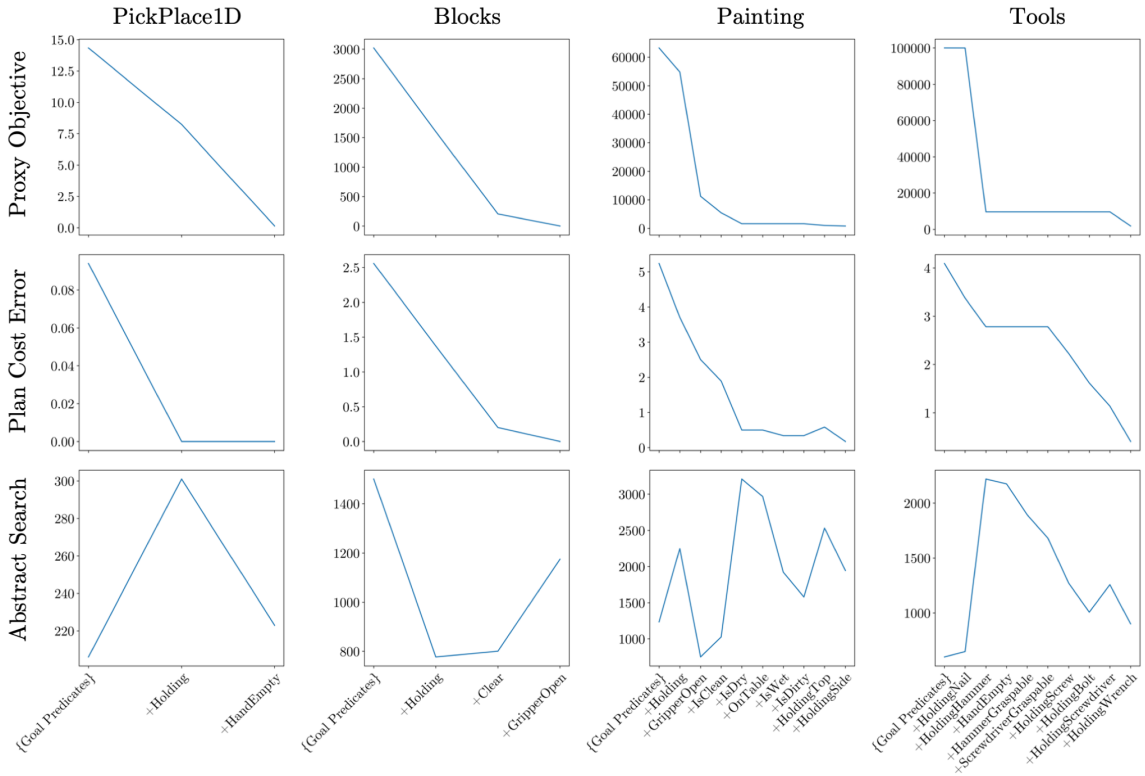


Figure 3-4: **Decomposing the proxy objective.** In these plots, each column corresponds to one environment. The x-axes correspond to sets of manually designed predicates. The predicate sets grow in size from left to right, starting with the goal predicates alone, adding one predicate at each tick mark, and concluding with the full set of manual predicates for the respective environment. The order that the predicates are added was determined by hill climbing with respect to the proxy objective. The top row shows the proxy objective itself; the middle row shows the plan cost error $|\textsc{Cost}(\hat{\pi}) - \textsc{Cost}(\pi^*)|$ minimized over the first 8 skeletons generated by abstract search; and the bottom row shows the total number of nodes created by the abstract search (our measure of abstract search time), cumulative over the 8 skeletons. *There are two key takeaways from this plot.* (1) The proxy objective (first row) monotonically decreases in all environments; this property makes local search over candidate predicate sets an attractive option. (2) Neither of the two components that make up the proxy objective — plan cost error (second row) or abstract search time (third row) — has the same monotonically decreasing property on its own, suggesting that both parts are necessary for making our predicate invention pipeline work. All results are means over 10 seeds.

```
P1() ≜ (∀ ?x:block . (?x.grasp <= -0.485))
P2(?y:target) ≜ (∀ ?x:block . ¬Covers(?x, ?y))
P3(?x:block) ≜ (∀ ?y:target . ¬Covers(?x, ?y))
P4(?x:block) ≜ ¬(?x.grasp <= -0.485)
P5() ≜ ¬(∀ ?x:block . (?x.grasp <= -0.485))
```

```
Op0:
Parameters: [?x0:block, ?x1:target]
Preconditions:
 P3(?x0:block)
 P4(?x0:block)
 P5()
 P2(?x1:target)
Add Effects:
 Covers(?x0:block, ?x1:target)
 P1()
Delete Effects:
 P3(?x0:block)
 P4(?x0:block)
 P5()
 P2(?x1:target)
Controller: (PickPlace, [])
```

```
Op1:
Parameters: [?x0:block]
Preconditions:
 P3(?x0:block)
 P1()
Add Effects:
 P4(?x0:block)
 P5()
Delete Effects:
 P1()
Controller: (PickPlace, [])
```

Figure 3-5: PickPlace1D learned abstractions (top: predicates, bottom: operators).

```
P1(?x:block) ≜ (?x.pose_z <= 0.875)
P2(?y:block) ≜ (∀ ?x:block . ¬On(?x, ?y))
P3(?x:block) ≜ ¬(?x.pose_z <= 0.875)
P4(?x:robot) ≜ ¬(?x.fingers <= 0.5)
```

```
Op0:
Parameters: [?x0:block, ?x1:robot]
Preconditions:
 P1(?x0:block)
 P4(?x1:robot)
 OnTable(?x0:block)
 P2(?x0:block)
Add Effects:
 P3(?x0:block)
Delete Effects:
 P1(?x0:block)
 P4(?x1:robot)
 OnTable(?x0:block)
Controller: (Pick, [?x1:robot,
                    ?x0:block])

Op1:
Parameters: [?x0:block, ?x1:block,
             ?x2:robot]
Preconditions:
 P3(?x1:block)
 P1(?x0:block)
 P2(?x0:block)
 P2(?x1:block)
Add Effects:
 P4(?x2:robot)
 P1(?x1:block)
 On(?x1:block, ?x0:block)
Delete Effects:
 P3(?x1:block)
 P2(?x0:block)
Controller: (Stack, [?x2:robot,
                     ?x0:block])

Op2:
Parameters: [?x0:block, ?x1:block,
             ?x2:robot]
Preconditions:
 P2(?x1:block)
 P1(?x1:block)
 P4(?x2:robot)
 P1(?x0:block)
 On(?x1:block, ?x0:block)
Add Effects:
 P3(?x1:block)
 P2(?x0:block)
Delete Effects:
 P4(?x2:robot)
 P1(?x1:block)
 On(?x1:block, ?x0:block)
Controller: (Pick, [?x2:robot,
                    ?x1:block])

Op3:
Parameters: [?x0:block, ?x1:robot]
Preconditions:
 P3(?x0:block)
 P2(?x0:block)
Add Effects:
 P1(?x0:block)
 P4(?x1:robot)
 OnTable(?x0:block)
Delete Effects:
 P3(?x0:block)
Controller: (PutOnTable, [?x1:robot])
```

Figure 3-6: Blocks learned abstractions (top: predicates, bottom: operators).

```
P1(?x:obj) ≜ (?x.color <= 0.125)
P2(?x:obj) ≜ (?x.dirtiness <= 0.498)
P3(?x:obj) ≜ (?x.grasp <= 0.25)
P4(?x:obj) ≜ (?x.pose_y <= -0.306)
P5(?x:obj) ≜ (?x.wetness <= 0.5)
P6() ≜ (∀ ?x:lid . (?x.is_open <= 0.5))
P7() ≜ (∀ ?x:obj, ?y:box . ¬InBox(?x, ?y))
P8() ≜ (∀ ?x:obj, ?y:box . ¬IsBoxColor(?x, ?y))
P9() ≜ (∀ ?x:obj . ¬(?x.grasp <= 0.25))
P10(?x:obj) ≜ (∀ ?y:shelf . IsShelfColor(?x, ?y))
P11(?x:obj) ≜ ¬(?x.color <= 0.125)
P12(?x:obj) ≜ ¬(?x.dirtiness <= 0.498)
P13(?x:obj) ≜ ¬(?x.grasp <= 0.5)
P14(?x:obj) ≜ ¬(?x.grasp <= 0.25)
P15(?x:obj) ≜ ¬(?x.held <= 0.5)
P16(?x:obj) ≜ ¬(?x.wetness <= 0.5)
P17(?x:robot) ≜ ¬(?x.fingers <= 0.5)
P18() ≜ ¬(∀ ?x:lid . (?x.is_open <= 0.5))
P19(?x:obj) ≜ ¬(∀ ?y:box . IsBoxColor(?x, ?y))
P20(?x:obj, ?y:box) ≜ ¬InBox(?x, ?y)
```

```
Op0:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P19(?x0:obj)
  P4(?x0:obj)
  P12(?x0:obj)
  P15(?x0:obj)
  P5(?x0:obj)
  P1(?x0:obj)
Add Effects:
  P16(?x0:obj)
  P2(?x0:obj)
Delete Effects:
  P12(?x0:obj)
  P5(?x0:obj)
Controller: (Wash, [?x1:robot])

Op1:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P16(?x0:obj)
  P19(?x0:obj)
  P2(?x0:obj)
  P4(?x0:obj)
  P15(?x0:obj)
  P1(?x0:obj)
Add Effects:
  P5(?x0:obj)
Delete Effects:
  P16(?x0:obj)
Controller: (Dry, [?x1:robot])
```

```
Op2:
Parameters: [?x0:obj, ?x1:shelf,
             ?x2:robot]
Preconditions:
  P19(?x0:obj)
  P2(?x0:obj)
  P4(?x0:obj)
  P15(?x0:obj)
  P5(?x0:obj)
  P1(?x0:obj)
Add Effects:
  P10(?x0:obj)
  IsShelfColor(?x0:obj, ?x1:shelf)
  P11(?x0:obj)
Delete Effects:
  P1(?x0:obj)
Controller: (Paint, [?x2:robot])

Op3:
Parameters: [?x0:lid, ?x1:robot]
Preconditions:
  P6()
  P17(?x1:robot)
  P9()
  P7()
Add Effects:
  P18()
Delete Effects:
  P6()
Controller: (OpenLid, [?x1:robot,
                       ?x0:lid])
```

Figure 3-7: Painting learned abstractions (top: predicates, bottom: operators part 1 of 2).

```
Op4:
Parameters: [?x0:obj, ?x1:shelf, ?x2:robot]
Preconditions:
  P3(?x0:obj)
  P19(?x0:obj)
  P2(?x0:obj)
  IsShelfColor(?x0:obj, ?x1:shelf)
  P11(?x0:obj)
  P4(?x0:obj)
  P15(?x0:obj)
  P10(?x0:obj)
  P5(?x0:obj)
Add Effects:
  InShelf(?x0:obj, ?x1:shelf)
  P17(?x2:robot)
  P9()
  P14(?x0:obj)
Delete Effects:
  P4(?x0:obj)
  P15(?x0:obj)
  P3(?x0:obj)
Controller: (Place, [?x2:robot])

Op5:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P14(?x0:obj)
  P18()
  P9()
  P7()
  P4(?x0:obj)
  P17(?x1:robot)
Add Effects:
  P15(?x0:obj)
  P13(?x0:obj)
Delete Effects:
  P17(?x1:robot)
Controller: (Pick, [?x1:robot, ?x0:obj])

Op6:
Parameters: [?x0:obj, ?x1:box, ?x2:robot]
Preconditions:
  P19(?x0:obj)
  P2(?x0:obj)
  P7()
  P4(?x0:obj)
  P8()
  P20(?x0:obj, ?x1:box)
  P15(?x0:obj)
  P5(?x0:obj)
  P1(?x0:obj)
Add Effects:
  P11(?x0:obj)
  IsBoxColor(?x0:obj, ?x1:box)
Delete Effects:
  P8()
  P19(?x0:obj)
  P1(?x0:obj)
Controller: (Paint, [?x2:robot])

Op7:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P14(?x0:obj)
  P9()
  P7()
  P13(?x0:obj)
  P4(?x0:obj)
  P15(?x0:obj)
Add Effects:
  P17(?x1:robot)
Delete Effects:
  P15(?x0:obj)
  P13(?x0:obj)
Controller: (Place, [?x1:robot])

Op8:
Parameters: [?x0:obj, ?x1:box, ?x2:robot]
Preconditions:
  P2(?x0:obj)
  P14(?x0:obj)
  P18()
  P11(?x0:obj)
  P9()
  P7()
  P13(?x0:obj)
  P4(?x0:obj)
  P20(?x0:obj, ?x1:box)
  IsBoxColor(?x0:obj, ?x1:box)
  P15(?x0:obj)
  P5(?x0:obj)
Add Effects:
  InBox(?x0:obj, ?x1:box)
  P17(?x2:robot)
Delete Effects:
  P7()
  P13(?x0:obj)
  P4(?x0:obj)
  P20(?x0:obj, ?x1:box)
  P15(?x0:obj)
Controller: (Place, [?x2:robot])

Op9:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P19(?x0:obj)
  P14(?x0:obj)
  P9()
  P4(?x0:obj)
  P17(?x1:robot)
Add Effects:
  P15(?x0:obj)
  P3(?x0:obj)
Delete Effects:
  P17(?x1:robot)
  P9()
  P14(?x0:obj)
Controller: (Pick, [?x1:robot, ?x0:obj])

Op10:
Parameters: [?x0:obj, ?x1:robot]
Preconditions:
  P4(?x0:obj)
  P15(?x0:obj)
  P3(?x0:obj)
  P7()
Add Effects:
  P17(?x1:robot)
  P9()
  P14(?x0:obj)
Delete Effects:
  P15(?x0:obj)
  P3(?x0:obj)
Controller: (Place, [?x1:robot])
```

Figure 3-8: Painting learned abstractions (operators part 2 of 2).

```
P1(?x:robot) ≜ (?x.fingers <= 0.5)
P2(?x:screwdriver) ≜ (?x.size <= 0.503)
P3() ≜ (∀ ?x:nail, ?y:contraption . NailPlaced(?x, ?y))
P4() ≜ (∀ ?x:screw, ?y:contraption . ¬ScrewPlaced(?x, ?y))
P5() ≜ (∀ ?x:screw . ¬ScrewFastened(?x))
P6(?x:bolt) ≜ ¬(?x.is_held <= 0.5)
P7(?x:hammer) ≜ ¬(?x.is_held <= 0.5)
P8(?x:nail) ≜ ¬(?x.is_held <= 0.5)
P9(?x:robot) ≜ ¬(?x.fingers <= 0.5)
P10(?x:screw) ≜ ¬(?x.is_held <= 0.5)
P11(?x:screwdriver) ≜ ¬(?x.is_held <= 0.5)
P12(?x:wrench) ≜ ¬(?x.is_held <= 0.5)
P13() ≜ ¬(∀ ?x:bolt . BoltFastened(?x))
P14() ≜ ¬(∀ ?x:bolt . ¬BoltFastened(?x))
P15() ≜ ¬(∀ ?x:nail, ?y:contraption . NailPlaced(?x, ?y))
P16() ≜ ¬(∀ ?x:screw, ?y:contraption . ¬ScrewPlaced(?x, ?y))
P17() ≜ ¬(∀ ?x:screw . ScrewFastened(?x))
P18(?x:bolt) ≜ ¬(∀ ?y:contraption . ¬BoltPlaced(?x, ?y))
```

```
Op0:
Parameters: [?x0:nail, ?x1:robot]
Preconditions:
 P9(?x1:robot)
 P15()
Add Effects:
 P12(?x0:nail)
 P1(?x1:robot)
Delete Effects:
 P9(?x1:robot)
Controller: (PickNail, [?x1:robot,
                        ?x0:nail])

Op1:
Parameters: [?x0:contraption,
             ?x1:nail, ?x2:robot]
Preconditions:
 P1(?x2:robot)
 P12(?x1:nail)
 P15()
Add Effects:
 P9(?x2:robot)
 NailPlaced(?x1:nail,
            ?x0:contraption)
Delete Effects:
 P1(?x2:robot)
 P12(?x1:nail)
Controller: (Place, [?x2:robot])

Op2:
Parameters: [?x0:hammer, ?x1:robot]
Preconditions:
 P9(?x1:robot)
 P15()
Add Effects:
 P1(?x1:robot)
 P12(?x0:hammer)
Delete Effects:
 P9(?x1:robot)
Controller: (PickHammer, [?x1:robot,
                          ?x0:hammer])

Op3:
Parameters: [?x0:contraption,
             ?x1:hammer, ?x2:nail,
             ?x3:robot]
Preconditions:
 NailPlaced(?x2:nail,
            ?x0:contraption)
 P12(?x1:hammer)
 P15()
 P1(?x3:robot)
Add Effects:
 NailFastened(?x2:nail)
Delete Effects:
Controller: (FastenNailWithHammer,
  [?x3:robot, ?x2:nail, ?x1:hammer,
   ?x0:contraption])
```

Figure 3-9: Tools learned abstractions (top: predicates, bottom: operators part 1 of 2).

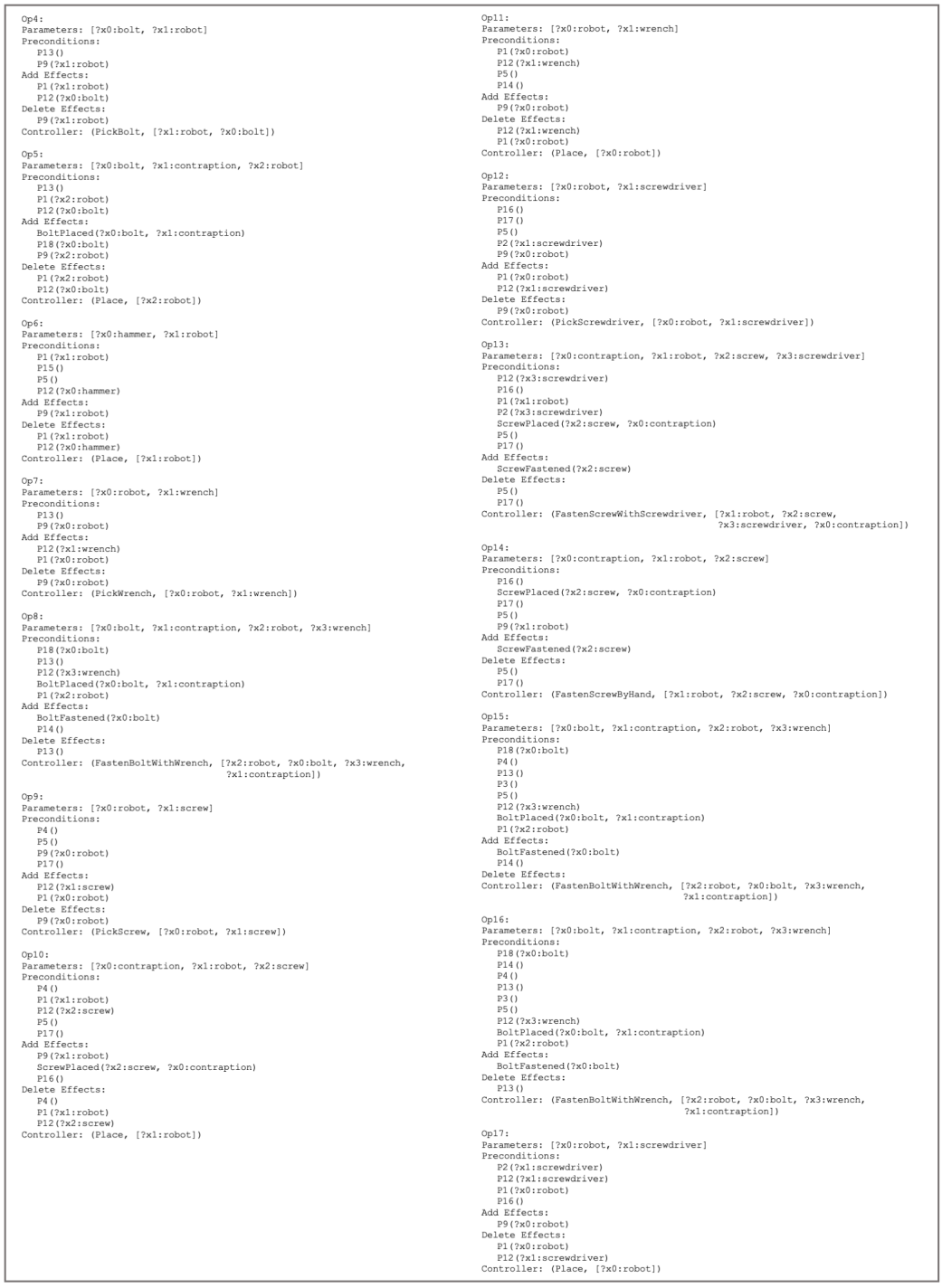


Figure 3-10: Tools learned abstractions (operators part 2 of 2).

Chapter 4

CAMPs: Learning Context-Specific Abstractions of Factored MDPs

4.1 Motivation

In this chapter and the remaining ones, we shift our focus from neuro-symbolic abstractions to projective abstractions, where the robot learns which aspects of a factored environment can be ignored during planning.

Online planning is a popular paradigm for sequential decision-making in robotics and beyond, but its practical application is limited by the computational burden of planning while performing a task. In *meta-planning*, the agent learns to guide planning efficiently and effectively based on its previous planning experience. *Learning to impose constraints* on the states considered and actions taken by an agent is a promising paradigm for meta-planning; it reduces the space of policies the agent must consider [82, 46, 150]. In contrast to (e.g., physical or kinematic) constraints beyond the agent's control, these constraints are imposed by the agent on itself for the sole purpose of efficient planning.

Beyond reducing the space of policies, imposing constraints can improve planning efficiency in another important way. In factored domains, where the states and actions decompose into variables, imposing constraints can induce *context-specific independences* (CSIs) [25] that render some variables irrelevant. For example, consider

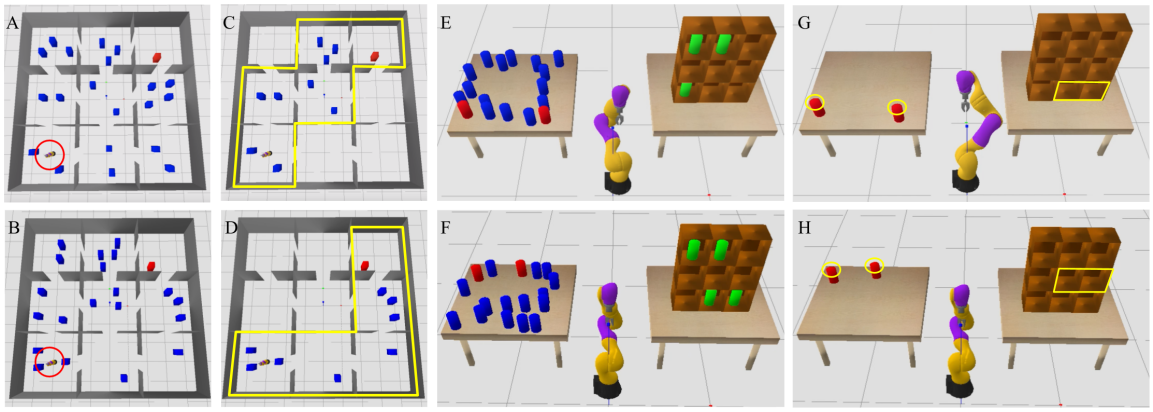


Figure 4-1: (A, B) In the NAMO domain, the robot (red circle) must reach the red object, which requires navigating there while moving obstacles out of the way. Two sample problems are shown. (C, D) If the robot constrains itself to stay within certain rooms (yellow), the obstacles in other rooms become irrelevant. (E, F) In the sequential manipulation domain, the robot must put the red objects into bins. (G, H) If the robot constrains itself to *top* grasps, the blue objects become irrelevant. Similarly, if the robot constrains target placements to certain bins (yellow), the green objects become irrelevant.

the two navigation among movable obstacles (NAMO) problems in Figure 4-1 (A, B). Imposing a constraint that forbids certain rooms induces CSIs between the robot's position and that of all obstacles in those forbidden rooms. Consequently, these obstacles can be ignored, as in Figure 4-1 (C, D). A planning problem with a context imposed[1] and the resulting irrelevant variables removed constitutes an *abstraction* of the original problem [98, 91]. Adopting the Markov decision process (MDP) formalism, we refer to this as a *context-specific abstract* MDP (CAMP).

Planning in a CAMP is often more efficient than planning in the original MDP, but may abstract away important details of the environment, leading to a suboptimal policy. Practically speaking, we are often interested in a trade-off: we would like our planners to produce highly rewarding behavior, but not be too computationally burdensome; we are willing to sacrifice optimality to maximize this trade-off in expectation. In this chapter, we propose a learning-based approach to maximize this trade-off. Given a set of training tasks with a shared transition model and factored states and actions, we first approximate the set of CSIs present in these tasks. We

[1]We henceforth use *context* as a synonym for *constraint*.

then train a *context selector*, which predicts a context that should be imposed for a given task. At test time, given a novel task, we use the learned context selector and CSIs to induce a CAMP, which we then use to plan. This overall pipeline is summarized in Figure 4-2.

Our approach rests on the premise that predicting contexts to impose is easier, and generalizes better, than learning a reactive policy. Intuitively, the burden on reactive policy learning is higher, as the policy must exactly carve out a specific, good path through transition space, whereas an imposed context must only carve out a region of transition space that includes at least one good path.

In experiments, we consider four domains, including robotic NAMO and sequential manipulation, that collectively exhibit discrete and continuous states and actions, relational states, sparse rewards, stochastic transitions, and long planning horizons. To evaluate the generality of CAMPs, we consider multiple planners, including Monte Carlo tree search [26], FastDownward [66], and a task and motion planner [139]. Our results suggest that planning with learned CAMPs strikes a strong balance between pure planning and pure policy learning [111]. In the NAMO domain, we also find that CAMPs with a generic task and motion planner outperform Stilman's NAMO-specific algorithm [142]. We conclude that CAMPs offer a promising path toward fast, effective planning in large and challenging domains.

4.2 Related Work

Our work falls under the broad research theme of learning to make planners more efficient using past planning experience. A fundamental question is deciding what to predict; for instance, it is common to learn a policy and/or value function from planning experience [134, 83, 82, 29]. By contrast, we learn to predict *contexts*. Recent work leverages a given set of contexts to represent planning problem instances in "score space" [82], but does not consider the resulting CSIs, which we showed experimentally to yield large performance improvements. Other methods predict the feasibility of task plans or motion plans [46, 150, 47], which can also be seen as

learning constraints on the search space. These methods can be readily incorporated into the CAMP framework.

We will formalize CAMPs as a particular class of MDP abstractions. There is a long line of work on deriving abstractions for MDPs, much of it motivated by the prospect of faster planning or more sample-efficient reinforcement learning [98, 68, 78, 141, 1]. One common technique is to *aggregate* states and actions into equivalence classes [17, 20, 91], a generalization of our notion of projective abstractions. Other work has learned to select abstractions [75]; a key benefit of CAMPs is that the contexts induce a structured hypothesis space of abstractions that greatly improve planning efficiency.

CAMPs identify and exploit CSIs [25] in factored planning problems. In graphical models, CSIs can be similarly used to speed up inference [156, 123, 45]. Stochastic Planning using Decision Diagrams (SPUDD) is a method that adapts these insights for planning with CSIs [70]. These insights are orthogonal to CAMPs, but could be integrated to yield further efficiencies. SPUDD is a pure planning approach that considers *all* contexts, whereas we *learn* a context selector that induces abstractions.

4.3 Problem Setting

A *task* is a pair of initial state and reward function, denoted $\omega = (s_0, R)$. We are given a set of N training tasks, $W_{\text{train}} = \{\omega^{(i)}\}_{i=1}^{N}$, and a test task, ω_{test}, all of which are drawn from some unseen distribution $P(\mathcal{W})$. In this chapter, we assume the reward is a function of state only, although this assumption is not critical to our methods and could be removed with minor changes. All tasks share the same factored state space $\mathcal{S}$, factored action space $\mathcal{A}$, transition model T, and horizon H; therefore, each task induces a finite-horizon factored MDP (Section 2.1), denoted $\mathcal{M}_\omega$. Each task is parameterized by a feature vector, denoted $\theta_\omega = \phi(s_0, R)$, with featurizer ϕ. For instance, in our robotic NAMO domain, ϕ gives a top-down image of the initial scene (which also implicitly describes the goal). The agent interacts with T and R as black boxes; it does not know their analytical representations or causal structure. We also assume a black-box MDP solver, PLAN, which takes as input an MDP $\mathcal{M}$ and

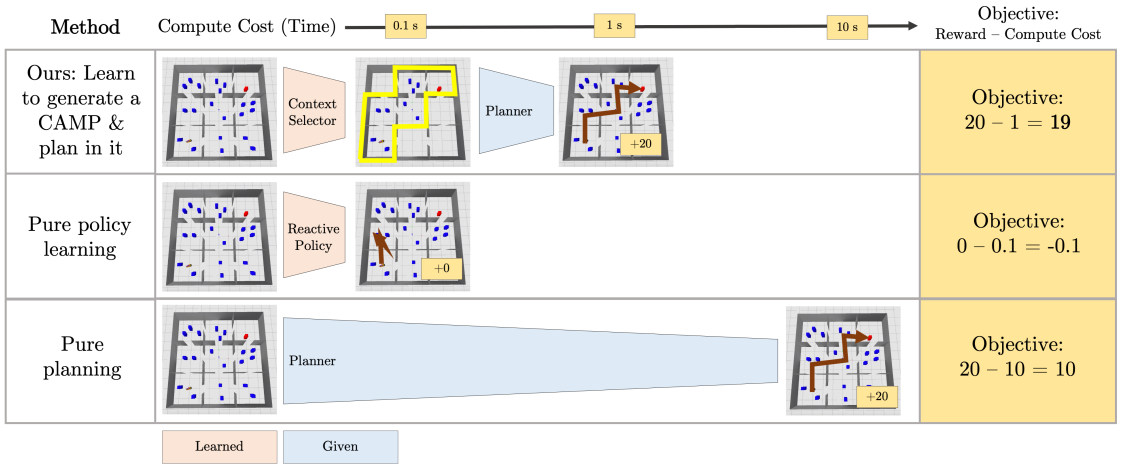


Figure 4-2: Three approaches to solving an MDP. Given a task, our approach (top row) applies its learned context selector to generate a CAMP, then plans in this CAMP to get a policy. Our approach often achieves higher reward than pure policy learning (middle row), and lower computational cost than pure planning (bottom row), leading to a good objective value (right; uses $\lambda = 1$ in Equation 4.1).

a current state s, and returns a next action: $a = \textsc{Plan}(\mathcal{M}, s)$.[2]

Before being presented with the test task, the agent may first interact with the training tasks W_{train}, perhaps compiling useful knowledge that it can deploy at test time, such as a task-conditioned policy. Then, it is given the test task $\omega_{\text{test}} = (s_{0,\text{test}}, R_{\text{test}})$, and its goal is to *efficiently* produce actions that accrue high cumulative reward in the test MDP $\mathcal{M}_{\omega_{\text{test}}}$. We formalize this trade-off via the objective:

$$J(\pi, \omega) = \mathbb{E}\left[\sum_{t=0}^{H} R(s_t) - \lambda \cdot \textsc{ComputeCost}(\pi, s_t)\right], \tag{4.1}$$

where $\omega = (s_0, R)$, $\textsc{ComputeCost}(\pi, s) \geq 0$ denotes the cost (e.g., wall-clock time) of evaluating the policy $\pi(s)$, $\lambda \geq 0$ is a trade-off parameter, and the expectation is over stochasticity in the transitions. Note that ComputeCost includes the cost of both computation performed before the agent starts acting and any computation that might be performed on each timestep after the first.

We seek to find $\pi_{\text{test}} = \text{argmax}_\pi J(\pi, \omega_{\text{test}})$. One possible approach is to call Plan

[2]Planners generally return a policy or a sequence of actions; we suppose that the planner is called at every timestep to simplify exposition. In our experiments, we replan in domains that have stochastic transitions.

on the full test MDP, that is, $\pi_{\text{test}}(s) = \text{PLAN}(\mathcal{M}_{\omega_{\text{test}}}, s)$. This method would yield high rewards, but it may also incur a large COMPUTECOST. Another possibility is to learn (at training time) and transfer (to test time) a task-conditioned reactive policy; this can have low COMPUTECOST at test time, but perhaps at the expense of rewards if the policy fails to generalize well to the test task (Figure 4-2).

4.4 Context-Specific Abstract Markov Decision Processes (CAMPs)

The objective formulated in Equation 4.1 trades off the computational cost of planning with the resulting rewards. This section presents an approach to optimizing this trade-off that lies between the two extremes of pure planning and pure policy learning [111]. Rather than planning in the full test task, we propose to *learn* to generate an abstraction [98, 91] of the test task, in which we can plan efficiently.

An *abstraction* over state space $\mathcal{S}$ and action space $\mathcal{A}$ is a pair of functions (σ, τ), with $\sigma : \mathcal{S} \mapsto \mathcal{S}'$ and $\tau : \mathcal{A} \mapsto \mathcal{A}'$, where $\mathcal{S}'$ and $\mathcal{A}'$ are *abstract* state and action spaces. We are specifically interested in abstractions that are projections: $\sigma([s^1, s^2, \ldots, s^n]) = [s^{i_1}, s^{i_2}, \ldots, s^{i_{n'}}]$ and $\tau([a^1, a^2, \ldots, a^m]) = [a^{j_1}, a^{j_2}, \ldots, a^{j_{m'}}]$. This has the effect of dropping $n - n'$ state variables and $m - m'$ action variables; the i and j superscripts refer respectively to the state and action variables that are not dropped.

The *relevant-variables projection* is a simple projective abstraction that has been studied in prior work (under different names) [16, 68, 23]. It drops all irrelevant variables, in the following sense:

Definition 8 (Variable relevance)**.** *Given a factored* MDP *with variables* V *and reward variables* V_{rew}*, any* $V^i \in V$ *is* relevant *iff* $\exists\ V^j \in V_{rew}$*,* $t \in \{0, \ldots, H\}$*, and* $t' \in \{t+1, \ldots, H\}$ *s.t.* $V_t^i \not\perp\!\!\!\perp V_{t'}^j$*.*

Intuitively, a variable is relevant if there is *any* possibility that its value at some timestep will have an eventual influence, directly or indirectly, on the value of the reward. Unfortunately, as identified by Baum et al. [16], relevance is often too strong

of a property for the relevant-variables projection to yield meaningful improvements in practice — most variables typically have *some* way of influencing the reward, under *some* sequence of actions taken by the agent. In search of greater flexibility, we now define a generalization of variable relevance that is conditioned on a particular context (Section 2.1.1).

Definition 9 (Context-specific variable relevance)**.** *Given a context* $(C, \mathcal{C})$ *and a factored* MDP *with variables* V *and reward variables* V_{rew}*, any* $V^i \in V \setminus C$ *is* relevant in the context $(C, \mathcal{C})$ *iff* $\exists\ V^j \in V_{rew}$*,* $t \in \{0, \ldots, H\}$*, and* $t' \in \{t+1, \ldots, H\}$ *s.t.* $V_t^i \not\perp\!\!\!\perp V_{t'}^j \mid (C, \mathcal{C})$.

Each possible context $(C, \mathcal{C})$ induces a projection that drops variables which are irrelevant in $(C, \mathcal{C})$; let $\text{proj}_{C,\mathcal{C}}$ denote this abstraction. We now define a CAMP, an abstract MDP associated with $\text{proj}_{C,\mathcal{C}}$.

Definition 10 (Context-Specific Abstract MDP (CAMP))**.** *Consider an* MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R, H)$ *and a context* $(C, \mathcal{C})$*. Let* $proj_{C,\mathcal{C}} = (\sigma, \tau)$ *with right inverses* (σ^{-1}, τ^{-1})*. Let* $\perp$ *be a new sink state, such that* $\perp \notin \sigma(\mathcal{S})$*. The context-specific abstract* MDP*,* $\mathcal{M}'$*, for* $\mathcal{M}$ *and* $(C, \mathcal{C})$ *is* $(\sigma(\mathcal{S}) \cup \{\perp\}, \tau(\mathcal{A}), T', R', H)$*, where* T' *and* R' *are defined as follows:* $\forall s'_t, s'_{t+1} \in \sigma(\mathcal{S}), a'_t \in \tau(\mathcal{A})$,

1. $T'(\perp, a'_t, \perp) = 1$
2. $T'(s'_t, a'_t, \perp) = 1$ *if* $(\sigma^{-1}(s'_t), \tau^{-1}(a'_t))$ *is not in the context;*
3. $T'(s'_t, a'_t, s'_{t+1}) = T(\sigma^{-1}(s'_t), \tau^{-1}(a'_t), \sigma^{-1}(s'_{t+1}))$ *if* $(\sigma^{-1}(s'_t), \tau^{-1}(a'_t))$ *is in the context;*
4. $R'(\perp) = -\infty$
5. $R'(s'_t) = R(\sigma^{-1}(s'_t))$

We may also say that $\mathcal{M}'$ *is* $\mathcal{M}$ with context $(C, \mathcal{C})$ imposed.

Intuitively, a CAMP imposes a projective abstraction that drops the variables that are irrelevant *under the given context*, and also imposes that any transition in violation

of the context leads the agent to $\perp$, an absorbing sink state with reward $-\infty$. In practice, the right inverses σ^{-1} and τ^{-1} can be obtained by assigning arbitrary values to the dropped variables; the choice of value is inconsequential by Definition 9. For a graphical example of a CAMP, see Figure 4-3.

A CAMP is usually *not* optimality-preserving, because the context restricts the agent to a subregion of the state and action space [2]. However, context-specific relevance is much weaker than relevance: it only requires a variable to be relevant under the given context. For example, to a robot operating in a home, the weather outside is irrelevant as long as it remains in the context of staying indoors.

CAMPs offer a way to solve the test task (Section 4.3) that lies between the extremes of pure planning and pure policy learning. Namely, given a test MDP $\mathcal{M}_{\omega_{\text{test}}}$, we select a context, compute the relevant variables under that context via backward induction, generate the CAMP, and finally plan in this CAMP to obtain a policy π_{test} for $\mathcal{M}_{\omega_{\text{test}}}$. We have therefore reduced the problem of optimizing $J(\pi, \omega_{\text{test}})$ to that of determining the best context $(C, \mathcal{C})$ to impose. To address this issue, we now turn to learning.

4.5 Learning to Generate CAMPs

We now have the ability to generate a context-specific abstract MDP (CAMP) when given a context and the associated context-specific independences. However, contexts and their associated independences are *not* provided in our problem. In this section, we describe how to learn approximate context-specific independences and a context selector, for use at test time. Figure 4-4 gives a data-flow diagram.

4.5.1 Approximating the Context-Specific Independences

Recall that the agent is given MDPs with factored states and variables, but only query access to the (shared) transition model. For example, the transition model may be a black-box physics simulator, as in two of our experimental domains. In order to approximately determine the context-specific independences that are latent

Input: State and action variables $V = \{S^1, \ldots, S^n\} \cup \{A^1, \ldots, A^m\}$
Input: Black-box transition model T
Input: Context $(C, \mathcal{C})$
Input: Number of samples k_1, k_2 `// Hyperparameters`
Returns: Approximate CSIs $\{(V^i, V^j) : V^j_{t+1} \perp\!\!\!\perp V^i_t \mid (C, \mathcal{C})\}$, for arbitrary t
`// Initialize all pairs of variables to be independent`
Initialize: CSIs $\leftarrow V \times V$
`// Sample` k_1 `state and action assignments in the context`
$U \leftarrow$ SAMPLEINCONTEXT$(V, C, \mathcal{C}, k_1)$
`// Test pairs of variables for dependence`
for $V^i, V^j \in V \setminus C$ **do**
  for $u \in U$ **do**
    for *up to* k_2 *samples* v^i *of* V^i **do**
      if $T(V^j_{t+1} \mid V_t = u) \neq T(V^j_{t+1} \mid V_t \setminus \{V^i_t\} = u_{-i}, V^i_t = v^i)$ **then**
        `//` V^j `is dependent on` V^i`; remove this pair from CSIs`
        CSIs $\leftarrow$ CSIs $\setminus \{(V^i, V^j)\}$
return CSIs

Algorithm 3: Pseudocode for learning context-specific independences (CSIs) for a given context. See Figure 4-3B for an example output, and see Section 4.5.1 for discussion.

within a factored MDP, we propose a sample-based procedure. Given a context, the algorithm examines each pair of state or action variables (V^i, V^j) and tests for empirical dependence, that is, whether any sampled value of V^i_t induces a change in the distribution of V^j_{t+1}, conditioned on the sampled values of the remaining variables. Pseudocode is provided in Algorithm 3.

The runtime of this algorithm depends on the size of the domain and the number of samples used to test dependence. In theory, the number of samples required to identify all independences could be arbitrarily large. In practice, for the tasks we considered in our experiments, including robotic manipulation and NAMO, we found this algorithm to be sufficient for detecting a useful set of context-specific independences. Moreover, our method is robust to errors in the discovered independences, which in the worst case will simply exclude some candidate abstractions from consideration.

Where Does the Space of Contexts Come From? Our approximate algorithm allows us to estimate independences *given a context*; this raises the question of which contexts should be evaluated. We propose a simple method for deriving a

space of possible contexts that works well across our varied experimental domains. From the set of variables V, we consider conjunctions and disjunctions up to some length (a hyperparameter), excluding any terms whose involved variables have joint domain size less than some threshold (another hyperparameter). Note that for any finite threshold, this procedure immediately excludes contexts involving continuous variables. While this family of contexts has the benefit of being fairly general to describe, we emphasize that other choices, e.g., more domain-specific context families, may be used as well [82, 47, 28]. Importantly, the space of considered contexts should always include the trivial universal context so that CAMPs can reduce to planning in the original problem when no useful abstractions are available.

4.5.2 Learning the Context Selector

The performance of a CAMP depends entirely on the selected context; if the context constrains the agent to a poor region of plan space, or induces independences that make important variables irrelevant, the resulting policy could get very low rewards. However, if the context is selected judiciously, the CAMP may exhibit substantial efficiency gains with minor impact on rewards.

We now describe an algorithm for learning to select a context that optimizes the objective (Equation 4.1). Pseudocode is provided in Algorithm 4. Given each training task $\omega^{(i)} \in W_{\text{train}}$, we first identify the best possible context $(C^{(i)}, \mathcal{C}^{(i)})^*$ according to the objective in Equation 4.1. This process sets up a supervised multiclass classification problem that maps the featurized representation of a task $\theta_{\omega^{(i)}}$ to the best context $(C^{(i)}, \mathcal{C}^{(i)})^*$ to impose on that task. We solve this classification problem by training a neural network with cross-entropy loss, resulting in a context selector $f_\alpha(\theta_\omega) = (C, \mathcal{C})^*$, where α denotes the parameters of the neural network. At test time, we choose a context by calling $f_\alpha(\theta_{\omega_{\text{test}}})$, generate the associated CAMP, and plan in this CAMP to efficiently obtain a policy for the test task.

Input: Training tasks $W_{\text{train}} = \{\omega^{(i)}\}_{i=1}^{N}$ with features $\{\theta_{\omega^{(i)}}\}_{i=1}^{N}$
Input: Black-box transition model T
Input: Set of contexts $\{(C, \mathcal{C})\}$
Input: All learned CSIs $(C, \mathcal{C}) \to \{(V^i, V^j) : V_{t+1}^j \perp\!\!\!\perp V_t^i \mid (C, \mathcal{C})\}$
Returns: Context selector $f_\alpha(\theta_\omega) = (C, \tilde{\mathcal{C}})^*$ `// Neural network with parameters` α
Initialize: Inputs for supervised learning $X \leftarrow [\theta_{\omega^{(1)}}, \ldots, \theta_{\omega^{(N)}}]$
Initialize: Targets for supervised learning $Y \leftarrow []$
for $\omega^{(i)} \in W_{train}$ **do**
    `// See Subroutine below`
    $Y[i] \leftarrow \text{argmax}_{(C,\mathcal{C})}$ ScoreContext$(\omega^{(i)}, (C, \mathcal{C}), T, \text{CSIs for } (C, \mathcal{C}))$
`// Perform supervised learning (multiclass classification)`
$\alpha^* \leftarrow \text{argmin}_\alpha$ CrossEntropyLoss$(X, Y; \alpha)$
return f_{α^*}

Subroutine ScoreContext
    Input: Training task $\omega = (s_0, R)$
    Input: Black-box transition model T
    Input: Context $(C, \mathcal{C})$
    Input: Learned CSIs $\{(V^i, V^j) : V_{t+1}^j \perp\!\!\!\perp V_t^i \mid (C, \mathcal{C})\}$
    Returns: A score
    $\mathcal{M}' \leftarrow$ CreateCAMP$(T, R, (C, \mathcal{C}), \text{CSIs})$ `// See Section 4.4`
    $\pi(s) \triangleq$ Plan$(\mathcal{M}', s)$ `// Plan in the` CAMP
    return $J(\pi, \omega)$ `// See Equation 4.1`

Algorithm 4: Pseudocode for learning a context selector model, given training tasks and their context-specific independences (CSIs). See Section 4.5.2 for discussion.

4.6 Experiments

Our experiments aim to answer the following key questions:

- How does planning with learned CAMPs compare to pure planning and pure policy learning across a varied set of domains, both discrete and continuous?
- To what extent is the performance of CAMPs planner-agnostic?
- How does the performance of CAMPs vary with the choice of λ (Equation 4.1)?
- How does the performance of CAMPs vary with the number of training tasks?

4.6.1 Experimental Design

4.6.1.1 Domains and Planners

We consider four domains and five planners (four online, one offline).

Domain D1: Gridworld. The first domain we consider is a simple maze-style gridworld in which the agent must navigate across rooms to reach a goal location, while avoiding obstacles that stochastically move around at each timestep. The agent has available to it REMOVE(OBJ) actions, which remove the given obstacle from the world so that the agent can no longer collide with it, but these actions can only be used when the agent is adjacent to the obstacle. Whenever the agent collides with an obstacle, it is placed back in its initial location. Each obstacle remains within a particular room, and so the agent can impose a context of not entering particular rooms, allowing it to ignore the obstacles that are in those rooms, and also not have to consider the action of removing those obstacles. Across tasks, we vary the maze layout. We train on 50 task instances and test on 10 held-out instances.

Planners. We consider the following planners for this domain: Monte Carlo tree search (MCTS), breadth-first graph search with replanning (BFSReplan), and asynchronous value iteration (VI). Both MCTS and BFSReplan are online planners, while VI is offline. As such, VI computes a policy over the full state space, and thus is only tractable in this relatively small (about 100,000 states) domain.

Representations. The features of each task are a top-down image of the maze layout. The state is a vector of the current position and room of each obstacle, the agent, and the goal. The actions are moving up, down, left, right; and removing each obstacle in the environment.

Domain D2: Classical planning. We next consider a deterministic classical planning domain in which an agent must make a meal for dinner, and has three options: to stay within the living room to make ramen, to go to the kitchen to make a sandwich, or to go to the store to buy and prepare a steak. Making any of these terminates the task. The steak gives higher terminal reward than the sandwich, which in turn gives higher terminal reward than the ramen. However, planning to go to the

store for steak requires reasoning about many objects that would be irrelevant under the context of staying within the home (for a sandwich or ramen), and planning to go to the kitchen for a sandwich requires reasoning about many objects that would be irrelevant under the context of staying within the living room (for ramen). There is also a timestep penalty, incentivizing the agent to finish quickly. Optimal plans may involve 2, 16, or 22 actions depending on the relative rewards for obtaining the ramen, sandwich, and steak. These rewards are the only thing that varies between task instances; there is thus small variation between task instances relative to the other domains. We train on 20 task instances and test on 25 held-out instances.

Planner. We use an off-the-shelf classical planner (Fast-Downward [66] in A^* mode with the *lmcut* heuristic). The various rewards are implemented as action costs. As this domain is deterministic, we only run the planner once per task; it is guaranteed to find a reward-maximizing trajectory.

Representations. The features of each task are a vector of the terminal rewards for each meal. The state is a binary vector describing which logical fluents hold true (1) versus false (0). The actions are logical operators described in PDDL, each containing parameters, preconditions, and effects.

Domain D3: Robotic navigation among movable obstacles (NAMO). Illustrated in Figure 4-1A, this domain has a robot navigating through rooms with the goal of reaching the red object in the upper-right room. Roughly 20 blue obstacles are scattered throughout the rooms, and like in the gridworld, the robot may impose the context of not entering particular rooms; it may also pick up the obstacles and move them out of its way. Across tasks, we vary the positions of all objects. We train on 50 task instances and test on 10 held-out instances. This domain has continuous states and actions, and as such is extremely challenging for planning. Though the obstacles do not move on their own (like they do in the gridworld), the difficulty of this domain stems from the added complexity of needing to reason about geometry and continuous trajectories. We simulate this domain using PyBullet [37]. The reward function is sparse: 1000 if the goal location is reached and 0 otherwise.

Planner. Developing planners for robotic domains with continuous states and

actions is an active area of research. For this domain, we use a state-of-the-art task and motion planner [139], which is *not* specific to NAMO problems. We use the RRT-Connect algorithm [93] for motion planning and the Fast-Forward PDDL planner [71] for task planning.

Representations. The features of each task are a top-down image of the scene. The state is a vector of the current pose of each object and the robot, and the robot's current room. The actions are moving the robot base to a target pose, and clearing an object in front of the robot.

Domain D4: Robotic sequential manipulation. Illustrated in Figure 4-1C, this domain has a robot manipulating the two red objects that start off on the left table to be placed into the bins on the right table. The fifteen blue objects on the left table serve as distractors, with which the robot must be careful not to collide when grasping the red objects; the green objects in the bins indicate that certain bins are already occupied. Across tasks, we vary the positions of all objects, and which bins are occupied by green objects. We also vary the radii of the red objects. We train on 50 task instances and test on 10 held-out instances. We again simulate this domain using PyBullet [37]. As in Domain D3, the reward function is sparse: 1000 if the goal location is reached and 0 otherwise.

Broadly, there are two types of contexts that are useful to impose in this domain. (1) If the robot chooses to constrain its *grasp style* to only allow top-grasping the red objects, then it need not worry about colliding with the blue objects, and can thus ignore them. However, this does not always work, since not all geometries are amenable to being top-grasped; for instance, sometimes an object's radius may be too large. Note, however, that to place the red objects into the bins upright, a side-grasp is necessary, and so we provide the robot a *regrasp* operator in addition to the standard move, pick, and place. Importantly, this regrasp operator is never *necessary*, but including it can allow the robot to simplify its planning problem by ignoring the blue objects (see Equation 4.1). (2) If the robot chooses to constrain which bins it will place the red objects into, then it need not worry about the green objects in the other bins, simplifying the planning problem.

Planner. Same as in Domain D3 (NAMO).

Representations. The features of each task are a vector of the object radii and occupied bins. The state is a vector of the current pose of each object, the grasp style used by the robot, and the current held object (if any). The actions are moving the robot to a target base pose and grasping at a target gripper pose (which requires an empty gripper), and moving the robot to a target base pose and placing at a target placement pose (which requires an object to be currently held).

4.6.1.2 Methods and Baselines

We consider the following methods:

- CAMP. Our full method.
- CAMP ablation. An ablation of our full method in which the CAMP only sends the agent to a sink state for context violation, but does *not* project away irrelevant variables.
- Pure planning. This baseline does not use the training tasks, and just solves the full test task.
- Plan transfer. This baseline solves each training task to obtain a plan, and at test time picks actions via majority vote across the training task plans.
- Policy learning. This baseline solves each training task to obtain a plan, then trains a state-conditioned neural network policy to imitate the resulting dataset of state-action trajectories, using supervised learning. This policy is used directly to choose actions at test time.
- Task-conditioned policy learning. This baseline is the same as policy learning, but the neural network also receives as input the features of the task, in addition to the current state.
- (Domain D3 only) Stilman's planning algorithm [142] for NAMO problems, named ResolveSpatialConstraints, which attempts to find a feasible path to a target location by first finding feasible paths to any obstructing objects and moving them out of the way.

In all our domains, every variable is relevant under *no* context. For this reason,

the pure planning baseline can also be understood as an ablation of CAMP that does not account for contexts.

4.6.1.3 Experimental Details

In all experiments, computational cost is measured in wall-clock time (seconds). We use the following values of λ: 0 for MCTS^3, 100 for BFSReplan, 250 for FastDownward, and 100 for TAMP. Every domain uses horizon $H = 25$. Additionally, to ensure that shorter plans are preferred in general, all domains use a discount factor $\gamma = 0.99$, except for Domain D2 which uses a timestep penalty as previously discussed.

To properly evaluate our objective (Equation 4.1), we would need to run every method until it completes, which can be extremely slow, e.g. for the pure planning baseline, or when our context selector picks a bad context. To safeguard against this, we impose a timeout of 60 seconds on all planning calls.

All neural networks are either fully connected for vector inputs or convolutional for image inputs. Fully connected networks have hidden layer sizes $\langle 50, 32, 10 \rangle$. Convolutional networks use a convolutional layer with 10 output channels and kernel size 2, followed by a max-pooling layer with stride 2, and then fully connected layers of [32, 10]. Neural networks are trained using the Adam optimizer [84] with learning rate 10^{-4}, until the loss on the training dataset reaches 10^{-3}.

To generate the spaces of contexts, we use the method described in Section 4.5.1. In Domains D1, D2, and D3, we consider disjunctive and single-term (a single variable and a single value in its domain) constraints only, while in Domain D4 we also consider conjunctive constraints. All contexts only consider the discrete variables in the domain. Our parameters k_1 and k_2 (Algorithm 3) are: $k_1 = k_2 = 50$ for Domain D1, $k_1 = k_2 = 40$ for Domain D2, and $k_1 = k_2 = 25$ for Domains D3 and D4.

Method	Test Task Objective Value (St. Dev.)				
	D1 (Grid), MCTS	D1 (Grid), BFSReplan	D2 (Classical)	D3 (NAMO)	D4 (Manip)
CAMP (ours)	**70** (16)	**21** (10)	-286 (9.6)	**896** (63)	**744** (94)
CAMP ablation	25 (11)	0.9 (24)	-308 (52)	707 (154)	453 (237)
Pure planning	6 (5)	-17 (11)	-414 (20)	242 (385)	335 (86)
Plan transfer	-7 (0.4)	-6 (15)	-467 (0.02)	141 (227)	21 (34)
Policy learning	-3 (4)	-11 (13)	-469 (0.2)	-0.2 (0.01)	-0.2 (0.01)
Task-conditioned	5 (5)	-2 (11)	**-145** (0.4)	-0.3 (0.01)	-0.2 (0.02)
Stilman's [142]	-	-	-	826 (36)	-

Table 4.1: Compilation of test task objective values on all our domains and methods. Objective values are computed using the same values of λ that were used during training. All table entries report an average over 10 independent runs of both context selector training and test task evaluation. Stilman's algorithm [142] is NAMO-specific and so is only run on the NAMO domain.

4.6.2 Main Results and Discussion

Table 4.1 shows the objective values (Equation 4.1) for all domains and methods. Figure 4-5 plots the mean returns versus computation time on the test tasks. All results report an average over 10 independent runs of context selector training and test task evaluation.

CAMPs outperform every baseline in all but Domain D2 (classical planning). CAMPs fare better than task-conditioned policy learning because the latter fails to generalize from training tasks to test tasks. This failure manifests in low test task rewards, and in a substantial difference between the training and test objective values. In classical planning, however, policy learning outperforms CAMPs; both achieve high task rewards, but the policy is faster to execute. This is because this domain involves little variation between instances, in stark contrast to the other domains.

Another clear conclusion from the main results is that CAMPs outperform pure planning across all experiments, consistently achieving lower computational costs. In several cases, including NAMO and manipulation, CAMPs also achieve higher *rewards* than pure planning does, since the latter sometimes hits the 60-second timeout before discovering the superior plans found very quickly by CAMPs.

Results for the CAMP ablation show that imposing contexts alone provides clear

[3]Since MCTS is an anytime algorithm, we give it a timeout of 0.25 seconds. With $\lambda = 0$, the objective then reflects the best returns found within this timeout.

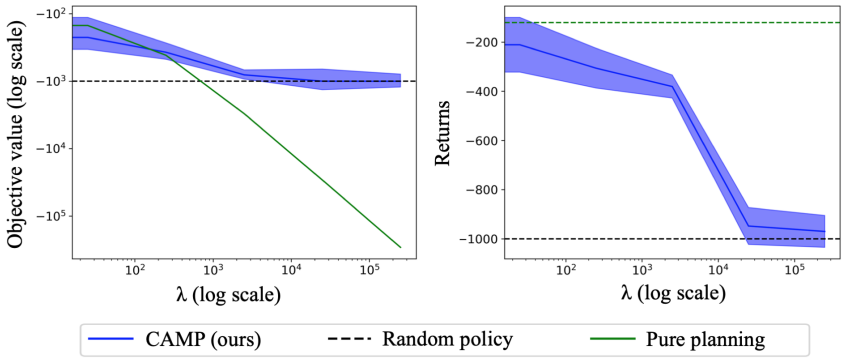
benefits, focusing the planner on a promising region of the search space. This result is consistent with prior work showing that learning to impose constraints reduces planning costs [82, 46, 150]. However, the difference between CAMP and the ablation shows that dropping irrelevant variables provides even greater benefits.

A final observation is that CAMPs perform comparably to Stilman's NAMO algorithm [142]. This is notable because Stilman's algorithm employs NAMO-specific assumptions, whereas the planner we use does not; in fact, we can see that the pure planner is strongly outperformed by Stilman's algorithm. In our method, the context selector learns to constrain the robot to stay in emptier rooms, meaning it must move comparatively few objects out of the way. This leads to efficient planning, making the computational cost of CAMPs almost as good as that of Stilman's algorithm; additionally, it leads CAMPs to obtain higher rewards than Stilman's algorithm, because it often reaches the goal faster. The NAMO results also show that CAMPs are able to learn to impose useful contexts even when there are multiple good options, e.g., multiple "room paths" with similar numbers of obstacles (Figure 4-1).

4.6.3 Additional Results

4.6.3.1 Performance as a Function of λ

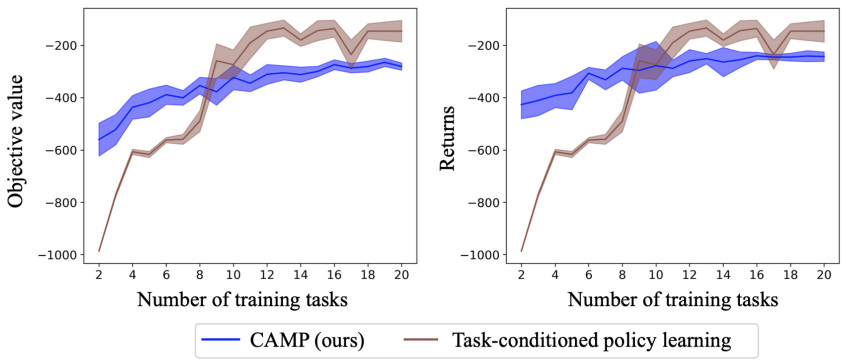
The plots on the right illustrate how the objective value (left) and returns (right) accrued by the CAMP policy vary as a function of λ, in Domain D2 (classical planning). The right one shows the returns from CAMP interpolate between those obtained by pure planning (when $\lambda = 0$, the agent is okay with spending a long time planning out its actions) and those obtained by a random



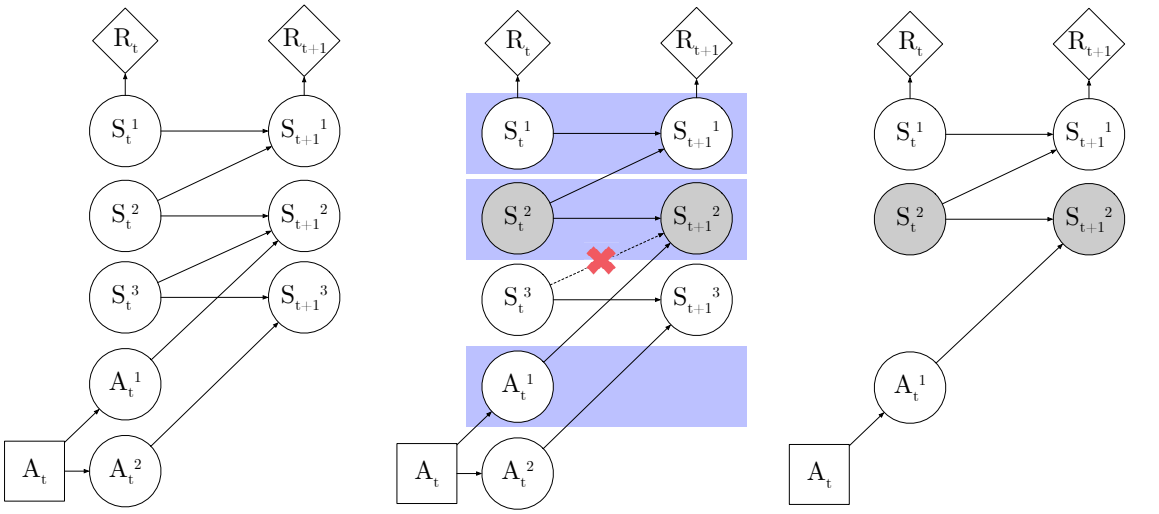
policy (when $\lambda \to \infty$, the agent spends as little time as possible choosing actions). The green line is dashed because pure planning does not use λ, so its returns are unaffected by the value of λ. The left one (note the log-scale y-axis) shows objective values. We see that CAMP never suffers a lower objective value than that of a random policy, while pure planning drops down greatly as λ increases. This is because as $\lambda \to \infty$, our context selector learns to choose contexts that induce very little planning (but get low returns).

4.6.3.2 Performance as a Function of Number of Training Tasks

The plots on the right illustrate how the objective value (left) and returns (right) accrued by the CAMP policy vary as a function of the number of training tasks, in Domain D2 (classical planning).



Discussion. As following a policy in the test task requires near-zero computational effort (our neural networks are small enough that inference is very fast), the red curves in both plots are nearly identical. Interestingly, in the regime of fewer training tasks (≤ 8), CAMP outperforms policy learning, despite policy learning performing better with the full set of 20 tasks. This leads us to believe that in domains where policy learning would perform well when given a lot of data, generating and planning in a CAMP may be a more viable strategy when data is limited. As shown in the main results, this disparity between CAMPs and policy learning is more dramatic for the other three domains, where task instances are far more varied, so much so that CAMPs sharply outperform policy learning for any reasonable number of training environments that we were able to test.



(A) Original MDP
all variables relevant

(B) MDP with Imposed Context
A^2 and S^3 now irrelevant

(C) Context-Specific Abstract MDP
irrelevant variables dropped

Figure 4-3: (A) Example of a factored MDP represented as an influence diagram [24]. As seen in the diagram, $V_{\text{rew}} = \{S^1\}$. With no contexts imposed, all variables are relevant. (B) Imposing contexts can induce new independences. In this example, a context involving S^2 is imposed, inducing an independence between S^2 and S^3 (red $\times$). Variables A^2 and S^3 are irrelevant under the imposed context; relevant variables are highlighted in blue. Note that relevance is a time-independent property. (C) Dropping the irrelevant variables leads to a CAMP, an abstraction of the original MDP.

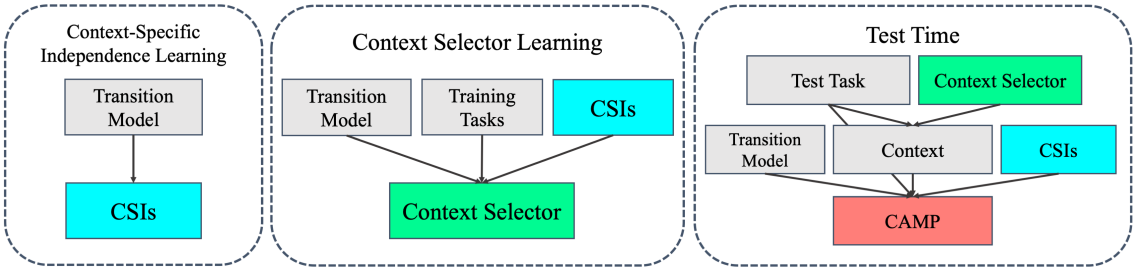


Figure 4-4: Data-flow diagram for our method during learning and test time. (Left) Approximate context-specific independence (CSI) learning derives the relevant state and action variables under each context. (Middle) A context selector is learned by optimizing the objective on the training tasks. The transition model and approximate CSIs are used to evaluate the objective. (Right) Given a test task, the agent selects a context to impose using the learned context selector. The relevant variables for this context are calculated from the learned CSIs. From the context, relevant variables, test task, and transition model, the agent derives a CAMP, and can plan in it to obtain a policy for the test task. Note that the contexts are used in a "one-shot" manner; there is no recourse if a bad one is selected.

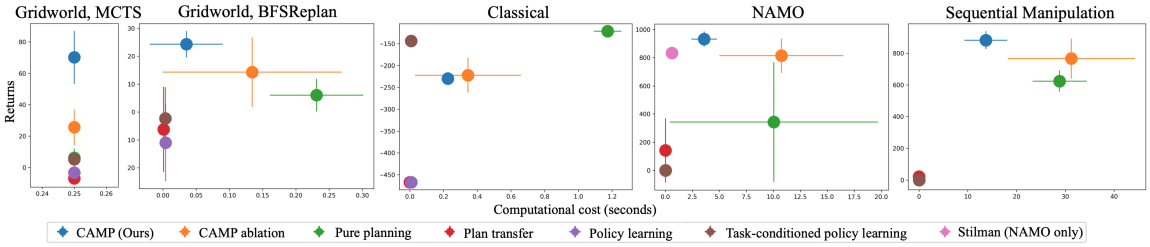


Figure 4-5: Mean returns versus computation time on the test tasks, for all domains and methods. All points report an average over 10 independent runs of training and evaluation, with lines showing per-axis standard deviations. CAMPs generally provide a better trade-off than the baselines: the blue points are usually higher than pure policy learning (CAMPs accrue more reward) and to the left of pure planning (CAMPs are more efficient). For the left-most plot, only the returns vary because MCTS is an anytime algorithm, so we run it with a fixed timeout. See Section 4.6.2 for discussion on these results.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Learning Compact Models for Planning with Exogenous Processes

5.1 Motivation

The previous chapter studied how a robot can learn projective abstractions by imposing constraints on its own behavior. However, what can we do if there are processes in the environment that are out of the robot's control (and therefore it cannot impose any constraints on those processes), some of which may be relevant to a given task? This chapter studies this problem through the lens of *exogenous* processes, which are processes that are unaffected by the robot's actions.

In fact, most aspects of the world are exogenous to any of us as individuals. However, though they are exogenous, these processes (e.g., weather and traffic) will often play a major role in the way we should choose to perform a task at hand. Despite being faced with such a daunting space of processes out of our control, humans are extremely adept at quickly reasoning about *which* aspects of this space to concern themselves with, for the particular task at hand.

Consider the setting of a household robot tasked with doing laundry inside a home. It should not get bogged down by reasoning about the current weather or traffic situation, because these factors are irrelevant to its task. If, instead, it were tasked with mowing the lawn, then good decision-making would require it to reason

about the time of day and weather (so it can finish the task by sunset, say).

In this chapter, we address the problem of approximately solving Markov decision processes (MDPs), without too much loss in solution quality, by leveraging the structure of their exogenous processes. We begin by formalizing the *mask-learning problem*. An autonomous agent is given a generative model of an MDP partitioned into an endogenous state (i.e., that which can be affected by the agent's actions) and a much higher-dimensional exogenous state. The agent must choose a *mask*, a subset of the exogenous state variables, that yields a policy *not too much worse* than a policy that would be obtained by reasoning about the entire exogenous state.

After formalizing the mask-learning problem, we discuss how we can leverage exogeneity to quickly learn transition models for only the relevant variables from data. Then, we explore the various value functions of interest within the problem, and discuss the conditions under which a policy for a particular mask will be optimal for the full MDP. Our analysis lends theoretical credence to the idea that a good mask should contain not only the exogenous state variables that directly influence the agent's reward function, but also ones whose dynamics are correlated (in the sense of mutual information) with theirs. This idea leads to a tractable approximate algorithm for the mask-learning problem that leverages the structure of the MDP, drawing upon mutual information among exogenous state variables.

We experiment in simulated robotic manipulation domains where a robot is put in a busy environment, among many other agents that also interact with the objects. We show that 1) in small domains where we can plan directly in the full MDP, the masks learned by our approximate algorithm yield competitive returns; 2) our approach outperforms strategies that do not leverage the structure of the MDP; and 3) our algorithm can scale up to planning problems with large exogenous state spaces.

5.2 Related Work

The notion of an *exogenous event*, one that is outside the agent's control, was first introduced in the context of planning problems by Boutilier et al. [24]. The work most

similar to ours is that by Dietterich et al. [43], who also consider the problem of model minimization in MDPs by removing exogenous state variables. Their formulation of an MDP with exogenous state variables is similar to ours, but the central assumption they make is that the reward decomposes additively into an endogenous state component and an exogenous state component. Under this assumption, the value function of the full MDP decomposes into two parts, and any policy that is optimal for the endogenous MDP is also optimal for the full MDP. On the other hand, we do not make this reward decomposition assumption, and so our value function does not decompose; instead, our work focuses on a different set of questions: 1) what are the conditions under which an optimal policy in a given reduced model is optimal in the full MDP? 2) can we build an algorithm that leverages exogeneity to efficiently (approximately) discover such a reduced model?

Model minimization of factored Markov decision processes is often defined using the notion of *stochastic bisimulation* [41, 55], which describes an equivalence relation among states based upon their transition dynamics. Other prior work in state abstraction tries to remove irrelevant state variables to form reduced models [78, 108]. Our approach differs from these techniques in two major ways: 1) we consider only reducing the exogenous portion of the state, allowing us to develop algorithms which leverage the computational benefits enjoyed by the exogeneity assumption; 2) rather than trying to build a reduced model that is faithful to the full MDP, we explicitly optimize a different objective (Equation 5.1), which tries to find a reduced model yielding high rewards in the full MDP. Recent work in model-free reinforcement learning has considered how to exploit exogenous events for better sample complexity [104, 34], whereas we tackle the problem from a model-based perspective.

If we view the state as a vector of features, then another perspective on our approach is that it is a technique for *feature selection* [63] applied to MDPs with exogenous state variables. This is closely related to the typical subset selection problem in supervised learning [87, 109, 77, 153], in which a learner must determine a subset of features that suffices for making predictions with little loss.

5.3 Problem Setting

In this section, we formalize the mask-learning problem for a factored MDP with exogenous variables.

We assume that the agent is given a *generative* model of an infinite-horizon factored MDP with exogenous variables (Section 2.1.2), in the sense of Kearns et al. [80]. Concretely, the agent is given the following:

- Knowledge of $\mathcal{S}$, $\mathcal{A}$, and γ.
- A black-box sampler of the transition model, which takes as input a state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, and returns a next state $s' \sim T(s, a, s')$.
- A black-box reward function, which takes as input a state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, and returns the reward $R(s, a)$ for that state-action pair.
- A black-box sampler of an initial state s_0 from some distribution $P(s_0)$.

We note that this assumption of having a generative model of an MDP lies somewhere in between that of only receiving execution traces (as in the typical reinforcement learning setting, assuming no ability to reset the environment) and that of having knowledge of the full analytical model. One can also view this assumption as saying that a simulator is available. Generative models are a natural way to specify a large MDP, as it is typically easier to produce samples from a transition model than to write out the complete next-state distributions.

We will focus on the setting where m, the number of exogenous state variables $x^1, x^2, \ldots, x^m$, is large. The fact that m is large precludes reasoning about the entire exogenous state. We will define the mask-learning problem as a discrete optimization problem for deciding which subset of variables $x^1, x^2, \ldots, x^m$ the agent should reason about.

To be able to unlink the effects of each exogenous state variable on the agent's reward, we will need to make an assumption on the form of the reward function $R(s_t, a_t) = R(n_t, x_t, a_t)$. This assumption is necessary for the agent to be able to reason about the effect of dropping a particular exogenous state variable from consideration. Specifically, we assume that $R(n_t, x_t, a_t) = \sum_{i=1}^{m} R^i(n_t, x_t^i, a_t)$. In words,

this says that the reward function decomposes into a sum over the individual effects of each exogenous state variable x^i. Although this means that the computation of the agent's reward cannot include logic based on any combinations of x^i, this assumption is not too restrictive: one could always construct "super-variables" encompassing state variables coupled in the reward. Note that despite this assumption, the value function might depend non-additively on the exogenous variables.

Our central problem of focus is that of finding a *mask*, a subset of the m exogenous state variables, that is "good enough" for planning, in the sense that we do not lose too much by ignoring the others. Before being able to formalize the problem, we must define precisely what it means to plan with only a subset of the m exogenous variables. Let $x = \begin{bmatrix} x^1 & x^2 & \dots & x^m \end{bmatrix}$, and $\tilde{x} \subseteq x$ be an arbitrary subset (a *mask*).

We define the *reduced model* $\tilde{M}$ corresponding to mask $\tilde{x}$ and MDP M as another MDP:

- $\mathcal{A}$ and γ are the same as in M.
- $\mathcal{S}$ is reduced by removing the dimensions corresponding to any $x^i \notin \tilde{x}$.
- $P(\tilde{s}_{t+1} \mid \tilde{s}_t, a_t) = P(n_{t+1} \mid n_t, a_t, \tilde{x}_t) \cdot P(\tilde{x}_{t+1} \mid \tilde{x}_t)$. Here, $\tilde{s}_t = \begin{bmatrix} n_t & \tilde{x}_t \end{bmatrix}$.[1]
- $R(\tilde{s}_t, a_t) = R(n_t, \tilde{x}_t, a_t) = \sum_{x^i \in \tilde{x}} R^i(n_t, x_t^i, a_t)$. Here, we are leveraging the assumption that the reward function decomposes as discussed above.

Since the agent only has access to a generative model of M, planning in $\tilde{M}$ will require estimating the reduced dynamics and reward models, $\hat{T}(\tilde{s}_t, a_t, \tilde{s}_{t+1})$ and $\hat{R}(\tilde{s}_t, a_t)$.

Formally, the *mask-learning problem* is to determine:

$$\tilde{x}^* = \operatorname*{argmax}_{\tilde{x} \subseteq x} J(\tilde{x}) = \operatorname*{argmax}_{\tilde{x} \subseteq x} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(n_t, x_t, \tilde{\pi}(n_t, \tilde{x}_t))\right] - \lambda \cdot \text{Cost}(\tilde{x}), \qquad (5.1)$$

where $\tilde{\pi}$ is the policy (mapping reduced states to actions) that is obtained by planning in $\tilde{M}$. In words, we seek the mask $\tilde{x}^*$ that yields a policy maximizing the expected

[1]The $\tilde{x}$ might, in actuality, not be Markov, since we are ignoring the variables $x \setminus \tilde{x}$. Nevertheless, this expression is well-formed, and estimating it from data marginalizes out these ignored variables. If they were not exogenous, such estimates would depend on the data-gathering policy (and thus could be very error-prone).

total reward *accrued in the actual environment* (the complete MDP M), minus a cost on $\tilde{x}$. Note that if $\lambda = 0$, then the choice $\tilde{x} = x$ is always optimal, and so λ serves as a regularization hyperparameter that balances the expected reward with the complexity of the considered mask. Reasonable choices of $\text{Cost}(\tilde{x})$ include $|\tilde{x}|$ or the amount of time needed to produce the policy $\tilde{\pi}$ corresponding to $\tilde{x}$.

5.4 Approach

5.4.1 Leveraging Exogeneity

The agent has only a generative model of the MDP M. In order to build a reduced MDP $\tilde{M}$, it must estimate $\hat{T}(\tilde{s}_t, a_t, \tilde{s}_{t+1}) = \hat{P}(n_{t+1} \mid n_t, a_t, \tilde{x}_t) \cdot \hat{P}(\tilde{x}_{t+1} \mid \tilde{x}_t)$ and $\hat{R}(\tilde{s}_t, a_t)$.

Recall that we are considering the setting where the space of exogenous variables is much larger than the space of endogenous variables. At first glance, then, it seems challenging to estimate $\hat{P}(\tilde{x}_{t+1} \mid \tilde{x}_t)$ using only the generative model for $P(x_{t+1} \mid x_t)$. However, this estimation problem is in fact greatly simplified due to the exogeneity of the x^i. To see why, consider the typical strategy for estimating a transition model from data: generate trajectories starting from some initial state x_0, then fit a one-step prediction model to this data. Now, if the x^i were endogenous, then we would need to commit to some policy π_{rollout} in order to generate these trajectories, and the reduced transition model we learn would depend heavily on this π_{rollout}. However, since the x^i are exogenous, we can roll out simulations conditioned *only on the initial state x_0, not on a policy*, and use this data to efficiently estimate the transition model $P(\tilde{x}_{t+1} \mid \tilde{x}_t)$ of the reduced MDP $\tilde{M}$.[2]

Because we can efficiently estimate the reduced model dynamics of exogenous state variables, we can not only tractably construct the reduced MDP, but also allow ourselves to explore algorithms that depend heavily on estimating these variables' dynamics, as we will do in Section 5.4.4.

[2]In high dimensions, we may still need a lot of data, unless the state factors nicely.

We now explore some value functions induced by the mask-learning problem, and use this analysis to develop a tractable but effective approximate algorithm for finding good masks.

5.4.2 Objective Estimation and Simple Strategies

Observe that the expectation in the objective $J(\tilde{x})$ (Equation 5.1) is the value $V_{\tilde{\pi}}(s_0)$ of an initial state under the policy $\tilde{\pi}$, corresponding to mask $\tilde{x}$. Because the full MDP M is very large, computing this value (the expected discounted sum of rewards) exactly will not be possible. Instead, we can use rollouts of $\tilde{\pi}$ to produce an estimate $\hat{V}_{\tilde{\pi}}$, which in turns yields an estimate of the objective, $\hat{J}(\tilde{x})$:

Procedure ESTIMATE-OBJECTIVE($M, \tilde{x}, n_{rollouts}$)

1 Construct estimated reduced MDP $\tilde{M}$ defined by mask $\tilde{x}$ and full MDP M.
2 Solve $\tilde{M}$ to get policy $\tilde{\pi}$.
3 **for** $i = 1, 2, \ldots, n_{rollouts}$ **do**
4 Execute $\tilde{\pi}$ in the full MDP M, obtain total discounted returns r_i.
5 $\hat{J}(\tilde{x}) = \frac{1}{n_{\text{rollouts}}} \sum_i r_i - \lambda \cdot \text{Cost}(\tilde{x})$

As discussed in Section 5.4.1, Line 1 is tractable due to the exogeneity of the x variables, which make up most of the dimensionality of the state. With ESTIMATE-OBJECTIVE in hand, we can write down some very simple strategies for solving the mask-learning problem:

- MASK-LEARNING-BRUTE-FORCE: Evaluate $\hat{J}(\tilde{x})$ for every possible mask $\tilde{x} \subseteq x$. Return the highest-scoring mask.
- MASK-LEARNING-GREEDY: Start with an empty mask $\tilde{x}$. While $\hat{J}(\tilde{x})$ increases, pick a variable x^i at random, add it into $\tilde{x}$, and re-evaluate $\hat{J}(\tilde{x})$.

While optimal, MASK-LEARNING-BRUTE-FORCE is of course intractable for even medium-sized values of m, as it will not be feasible to evaluate all 2^m possible subsets of x. Unfortunately, even MASK-LEARNING-GREEDY will likely be ineffective for medium and large MDPs, as it does not leverage the structure of the MDP whatsoever. To develop a better algorithm, we will start by exploring the connection between the value functions of the reduced MDP $\tilde{M}$ and the full MDP M.

5.4.3 Analyzing the Value Functions of Interest

It is illuminating to outline the various value functions at play within our problem. We have:

- $V^*(s)$: the value function under an optimal policy; unknown and difficult to compute exactly.
- $V_{\tilde{\pi}}(s)$: given a mask $\tilde{x}$, the value function under the policy $\tilde{\pi}$; unknown and difficult to compute exactly. Note that $V^*(s) \geq V_{\tilde{\pi}}(s)\ \forall s \in \mathcal{S}$, by definition.
- $\hat{V}_{\tilde{\pi}}(s)$: given a mask $\tilde{x}$, the empirical estimate of $V_{\tilde{\pi}}(s)$, which the agent can obtain by rolling out $\tilde{\pi}$ in the environment many times, as was done in the ESTIMATE-OBJECTIVE procedure.
- $\tilde{V}_{\tilde{\pi}}(\tilde{s})$: given a mask $\tilde{x}$, the value function of policy $\tilde{\pi}$ *within the reduced* MDP $\tilde{M}$. Here, $\tilde{s}$ is the reduced form of state s (i.e., the endogenous state n concatenated with $\tilde{x}$).

Intuitively, $\tilde{V}_{\tilde{\pi}}(\tilde{s})$ corresponds to the expected reward that the agent *believes* it will receive by following $\tilde{\pi}$, which typically will not match $V_{\tilde{\pi}}(s)$, the *actual* expected reward. In general, we cannot say anything about the ordering between $\tilde{V}_{\tilde{\pi}}(\tilde{s})$ and $V_{\tilde{\pi}}(s)$. For instance, if the mask $\tilde{x}$ ignores some negative effect in the environment, then the agent will expect to receive *higher* reward than it actually receives during its rollouts. On the other hand, if the mask $\tilde{x}$ ignores some positive effect in the world, then the agent will expect to receive *lower* reward than it actually receives.

It is now natural to ask: under what conditions would an optimal policy $\tilde{\pi}$ for the reduced MDP $\tilde{M}$ also be optimal for the full MDP M? The following theorem describes sufficient conditions:

Theorem 1. *Consider an* MDP M *as defined in Section 5.3, with exogenous state variables* $x = \begin{bmatrix} x^1 & x^2 & \dots & x^m \end{bmatrix}$, *and a mask* $\tilde{x} \subseteq x$. *Let* $\bar{x} = x \setminus \tilde{x}$ *be the variables not included in the mask. If the following conditions hold: (1)* $R^i(n_t, x_t^i, a_t) = 0\ \forall x^i \in \bar{x}$, *(2)* $P(n_{t+1} \mid n_t, a_t, x_t) = P(n_{t+1} \mid n_t, a_t, \tilde{x}_t)$, *(3)* $P(\tilde{x}_{t+1}, \bar{x}_{t+1} \mid \tilde{x}_t, \bar{x}_t) = P(\tilde{x}_{t+1} \mid \tilde{x}_t) \cdot P(\bar{x}_{t+1} \mid \bar{x}_t)$; *then* $\tilde{V}_{\tilde{\pi}}(\tilde{s}) = V_{\tilde{\pi}}(s)\ \forall s \in \mathcal{S}$. *If* $\tilde{\pi}$ *is optimal for the reduced* MDP $\tilde{M}$, *then it must also be true that* $\tilde{V}_{\tilde{\pi}}(\tilde{s}) = V^*(s)\ \forall s \in \mathcal{S}$.

Proof: Consider an arbitrary state $s \in \mathcal{S}$, with corresponding reduced state $\tilde{s}$. We begin by showing that under the stated conditions, $\tilde{V}_{\tilde{\pi}}(\tilde{s}) = V_{\tilde{\pi}}(s)$. The recursive form of these value functions is:

$$V_{\tilde{\pi}}(s) = R(s, \tilde{\pi}(\tilde{s})) + \gamma \sum_{s'} P(s' \mid s, \tilde{\pi}(\tilde{s})) \cdot V_{\tilde{\pi}}(s'),$$

$$\tilde{V}_{\tilde{\pi}}(\tilde{s}) = R(\tilde{s}, \tilde{\pi}(\tilde{s})) + \gamma \sum_{\tilde{s'}} P(\tilde{s'} \mid \tilde{s}, \tilde{\pi}(\tilde{s})) \cdot \tilde{V}_{\tilde{\pi}}(\tilde{s'}).$$

Now, consider an iterative procedure for obtaining these value functions, which repeatedly applies the above equations starting from $V^0_{\tilde{\pi}}(s) = \tilde{V}^0_{\tilde{\pi}}(\tilde{s}) = 0 \; \forall s \in \mathcal{S}$. Let the value functions at iteration k be denoted as $V^k_{\tilde{\pi}}(s)$ and $\tilde{V}^k_{\tilde{\pi}}(\tilde{s})$. We will show by induction on k that $V^k_{\tilde{\pi}}(s) = \tilde{V}^k_{\tilde{\pi}}(\tilde{s}) \; \forall k$.

The base case is immediate. Suppose $V^k_{\tilde{\pi}}(s) = \tilde{V}^k_{\tilde{\pi}}(\tilde{s}) \; \forall s \in \mathcal{S}$, for some value of k.

We compute:

$$
\begin{aligned}
V_{\tilde{\pi}}^{k+1}(s) &= R(s, \tilde{\pi}(\tilde{s})) + \gamma \sum_{s'} P(s' \mid s, \tilde{\pi}(\tilde{s})) \cdot V_{\tilde{\pi}}^{k}(s') \\
&= \sum_{i=1}^{m} R^i(n, x^i, \tilde{\pi}(\tilde{s})) + \gamma \sum_{s'} P(n' \mid n, \tilde{\pi}(\tilde{s}), x) \cdot P(x' \mid x) \cdot V_{\tilde{\pi}}^{k}(s') \\
&= \sum_{x^i \in \tilde{x}} R^i(n, x^i, \tilde{\pi}(\tilde{s})) + \gamma \sum_{s'} P(n' \mid n, \tilde{\pi}(\tilde{s}), x) \cdot P(x' \mid x) \cdot V_{\tilde{\pi}}^{k}(s') \\
&= R(\tilde{s}, \tilde{\pi}(\tilde{s})) + \gamma \sum_{s'} P(n' \mid n, \tilde{\pi}(\tilde{s}), x) \cdot P(x' \mid x) \cdot V_{\tilde{\pi}}^{k}(s') \\
&= R(\tilde{s}, \tilde{\pi}(\tilde{s})) + \gamma \sum_{s'} P(n' \mid n, \tilde{\pi}(\tilde{s}), \tilde{x}) \cdot P(x' \mid x) \cdot V_{\tilde{\pi}}^{k}(s') \\
&= R(\tilde{s}, \tilde{\pi}(\tilde{s})) + \gamma \sum_{n', \tilde{x'}} \sum_{\bar{x'}} P(n' \mid n, \tilde{\pi}(\tilde{s}), \tilde{x}) \cdot P(\tilde{x'}, \bar{x'} \mid \tilde{x}, \bar{x}) \cdot V_{\tilde{\pi}}^{k}(s') \\
&= R(\tilde{s}, \tilde{\pi}(\tilde{s})) + \gamma \sum_{n', \tilde{x'}} \sum_{\bar{x'}} P(n' \mid n, \tilde{\pi}(\tilde{s}), \tilde{x}) \cdot P(\tilde{x'} \mid \tilde{x}) \cdot P(\bar{x'} \mid \bar{x}) \cdot V_{\tilde{\pi}}^{k}(s') \\
&= R(\tilde{s}, \tilde{\pi}(\tilde{s})) + \gamma \sum_{n', \tilde{x'}} \sum_{\bar{x'}} P(n' \mid n, \tilde{\pi}(\tilde{s}), \tilde{x}) \cdot P(\tilde{x'} \mid \tilde{x}) \cdot P(\bar{x'} \mid \bar{x}) \cdot \tilde{V}_{\tilde{\pi}}^{k}(n', \tilde{x'}) \\
&= R(\tilde{s}, \tilde{\pi}(\tilde{s})) + \gamma \sum_{n', \tilde{x'}} P(n' \mid n, \tilde{\pi}(\tilde{s}), \tilde{x}) \cdot P(\tilde{x'} \mid \tilde{x}) \cdot \tilde{V}_{\tilde{\pi}}^{k}(n', \tilde{x'}) \cdot \cancel{\sum_{\bar{x'}} P(\bar{x'} \mid \bar{x})} \\
&= R(\tilde{s}, \tilde{\pi}(\tilde{s})) + \gamma \sum_{\tilde{s'}} P(\tilde{s'} \mid \tilde{s}, \tilde{\pi}(\tilde{s})) \cdot \tilde{V}_{\tilde{\pi}}^{k}(\tilde{s'}) = \tilde{V}_{\tilde{\pi}}^{k+1}(\tilde{s}).
\end{aligned}
$$

We have shown that $V_{\tilde{\pi}}^{k}(s) = \tilde{V}_{\tilde{\pi}}^{k}(\tilde{s})\ \forall k$. By standard arguments [124], this iterative procedure converges to the true $V_{\tilde{\pi}}(s)$ and $\tilde{V}_{\tilde{\pi}}(\tilde{s})$ respectively. Therefore, we have that $\tilde{V}_{\tilde{\pi}}(\tilde{s}) = V_{\tilde{\pi}}(s)$.

Now, if $\tilde{\pi}$ is optimal for $\tilde{M}$, then it is optimal for the full MDP M as well. This is because Condition (1) assures us that the variables not considered in the mask do not affect the reward, implying that $\tilde{\pi}$ optimizes the expected reward in not just $\tilde{M}$, but also M. Therefore, under this assumption, we have that $\tilde{V}_{\tilde{\pi}}(\tilde{s}) = V_{\tilde{\pi}}(s) = V^*(s)\ \forall s \in \mathcal{S}$. □

The conditions in Theorem 1 are very intuitive: (1) all exogenous variables *not* in the mask do not influence the agent's reward, (2) the endogenous state transitions do not depend on variables not in the mask, and (3) the variables in the mask transition

independently of the ones not in the mask. If these conditions hold, and we use an optimal planner for the reduced model, then we will obtain a policy that is optimal *for the full* MDP*, not just the reduced one.* Based on these conditions, it is clear that our mask-learning algorithm should be informed by two things: 1) the agent's reward function and 2) the degree of correlation among the dynamics of the various state variables.

Hoeffding's inequality [69] allows us to bound the difference between $V_{\tilde{\pi}}(s)$ and the empirical estimate $\hat{V}_{\tilde{\pi}}(s)$. For rewards in the range $[0, R_{\max}]$, discount factor γ, number of rollouts n, and policy $\tilde{\pi}$, we have that for any state s, $|V_{\tilde{\pi}}(s) - \hat{V}_{\tilde{\pi}}(s)| \leq \lambda$ with probability at least $1 - 2e^{-2n\lambda^2(1-\gamma)^2/R_{\max}^2}$. This justifies the use of $\hat{V}_{\tilde{\pi}}(s)$ in place of $V_{\tilde{\pi}}(s)$, as was done in the ESTIMATE-OBJECTIVE procedure. Next, we describe a tractable algorithm for mask-learning that draws on Theorem 1.

5.4.4 A Correlational Algorithm for Mask-Learning

Of course, it is very challenging to directly search for a low-cost mask $\tilde{x}$ that meets all the conditions of Theorem 1, if one even exists. However, we can use the intuition of those conditions to develop an approximate algorithm based upon greedy forward selection techniques for feature selection [63], which at each iteration add a single variable that most improves some performance metric. See Algorithm 5 for pseudocode.

Algorithm description. In line with Condition (1) of Theorem 1, we begin by estimating the set of exogenous state variables relevant to the agent's reward function, which we can do by leveraging the generative model of the MDP. As shown by the ESTIMATE-REWARD-VARIABLES subroutine, the idea is to compute, for each variable x^i, the average amount of variance in the reward across different values of x^i when the remainder of the state is held fixed, and threshold this at τ_{variance}. We define our initial mask to be this set. The efficiency-accuracy tradeoff of this subroutine can be controlled by tuning the number of samples, n_1 and n_2. In practice, we can make a further improvement by heuristically biasing the sampling to favor states that are

Algorithm MASK-LEARNING-CORRELATIONAL($M, \tau_{correl}, \tau_{variance}, n_1, n_2$)

$\tilde{x} \leftarrow$ ESTIMATE-REWARD-VARIABLES($M, \tau_{\text{variance}}, n_1, n_2$) `// Initial mask for MDP` M.

while $\hat{J}(\tilde{x})$ *increases* **do** `// Iteratively add to initial mask.`

  $x^i = \text{argmax}_{x^j \notin \tilde{x}} D_{\text{KL}}(\hat{T}_{\tilde{s},x^j} \parallel \hat{T}_{\tilde{s}} \otimes \hat{T}_{x^j})$ `// Measure mutual informations.`

  if $D_{KL}(\hat{T}_{\tilde{s},x^i} \parallel \hat{T}_{\tilde{s}} \otimes \hat{T}_{x^i}) < \tau_{correl}$ **then**

    break `// All mutual informations below threshold, terminate.`

  $\tilde{x} = \tilde{x} \cup x^i$

  $\hat{J}(\tilde{x}) \leftarrow$ESTIMATE-OBJECTIVE($M, \tilde{x}$)

return $\tilde{x}$

Subroutine ESTIMATE-REWARD-VARIABLES($M, \tau_{variance}, n_1, n_2$)

for *each variable* $x^i \in x$ **do**

  for n_1 *random samples of endog. state, exog. variables* $x \setminus x^i$*, action* a **do**

    Randomly sample n_2 settings of variable x^i, construct full states $s_1, s_2, \ldots, s_{n_2}$.

    Compute $\sigma_j^2 \leftarrow \text{Var}(\{R(s_1, a), \ldots, R(s_{n_2}, a)\})$.

  if $\frac{1}{n_1} \sum_{j=1}^{n_1} \sigma_j^2 > \tau_{variance}$ **then** `// Mean variance above threshold, accept.`

    emit x^i

Algorithm 5: Correlational mask learning. See Section 5.4.4 for details.

more likely to be encountered by the agent (such heuristics can be computed, for instance, by planning in a relaxed version of the problem).

Then, we employ an iterative procedure to approximately build toward Conditions (2) and (3): that the variables not in the mask transition independently of those in the mask and the endogenous state. To do so, we greedily add variables into the mask based upon the mutual information between the empirical transition model of the reduced state and that of each remaining variable. Intuitively, this quantitatively measures: "*How much better would we be able to predict the dynamics of* $\tilde{s}$ *if* $\tilde{s}$ *included variable* x^i*, versus if it didn't?*" This mutual information will be 0 if x^i transitions independently of all variables in $\tilde{s}$. To calculate the mutual information between $\tilde{s}$ and variable x^i, we must first learn the empirical transition models $\hat{T}_{\tilde{s},x^i}$, $\hat{T}_{\tilde{s}}$, and $\hat{T}_{x^i}$ from data. The exogeneity of most of the state is critical here: not only does it make

learning these models much more efficient, but also, without exogeneity, we could not be sure whether two variables actually transition independently of each other or we just happened to follow a data-gathering policy that led us to believe so.

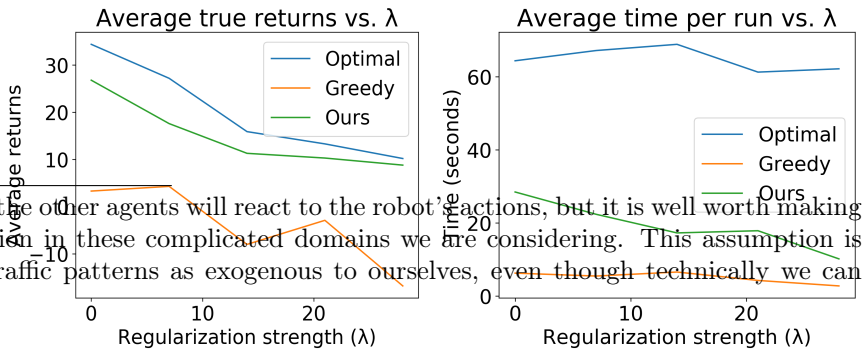
5.5 Experiments

Our experiments are designed to answer the following questions: (1) In small domains, are the masks learned by our algorithm competitive with the optimal masks? (2) Quantitatively, how well do the learned masks perform in large, complicated domains? (3) Qualitatively, do the learned masks properly reflect different goals given to the robot? (4) What are the limitations of our approach?

We experiment in simulated robotic manipulation domains in which a robot is placed in a busy environment with objects on tables, among many other agents that are also interacting with objects. The robot is rewarded for navigating to a given goal object (which changes on each episode) and penalized for crashing into other agents. The exogenous variables are the states of the other agents,[3] a binary-valued occupancy grid discretizing the environment, the object placements on tables, and information specifying the goal. We plan using value iteration with a timeout of 60 seconds. Empirical value estimates are computed as averages across 500 independent rollouts. Each result reports an average across 50 independent trials. We regularize masks by setting $\text{Cost}(\tilde{x}) = |\tilde{x}|$; our initial experimentation suggested that other mask choices, such as planning time, perform similarly.

In small domains, are our learned masks competitive with the optimal masks?

Finding the optimal mask $\tilde{x}^*$ is only possible in small



[3]It is true that typically, the other agents will react to the robot's actions, but it is well worth making the exogeneity assumption in these complicated domains we are considering. This assumption is akin to how we treat traffic patterns as exogenous to ourselves, even though technically we can slightly affect them.

models. Therefore, to answer this question we constructed a small gridworld representation of our experimental domain, with only ∼600 states and 5 exogenous variables, in which we can plan exactly. We find $\tilde{x}^*$ using the MASK-LEARNING-BRUTE-FORCE strategy given in Section 5.4.2. We compare the optimal mask with both 1) *Ours*, the mask returned by our algorithm in Section 5.4.4; and 2) *Greedy*, the mode mask chosen across 10 independent trials of the MASK-LEARNING-GREEDY strategy given in Section 5.4.2.

Discussion. For higher values of λ especially, our algorithm performs on par with the optimal brute-force strategy, which is only viable in small domains. The gap widens as λ decreases because as the optimal mask gets larger, it becomes harder to find using a forward selection strategy such as ours. In practical settings, one should typically set λ quite high so that smaller masks are preferred, as these will yield the most compact reduced models. Meanwhile, the greedy strategy does not perform well because it disregards the structure of the MDP and the conditions of Theorem 1. We also observe that our algorithm takes significantly less time than the optimal brute-force strategy; we should expect this gap to widen further in larger domains. Our algorithm's stopping condition is that the score function begins to decrease, and so it tends to terminate more quickly for higher values of λ, as smaller masks become increasingly preferred.

Quantitatively, how well do the learned masks perform in large, complicated domains?

To answer this question, we consider two large domains, visualized in Figure 5-1 (bottom row): *Factory*, in which a robot must fulfill manipulation tasks being issued in a stream; and *Crowd*, in which a robot must navigate to target objects among a crowd of agents executing their own fixed stochastic policies. Even after discretization, these domains have 10^{11} and 10^{80} states respectively; *Factory* has 22 exogenous variables and *Crowd* has 124. In either domain, both planning exactly

Algorithm	Domain	Average True Returns	Time / Run (sec)
Greedy	Factory	0	13.4
Ours (first phase)	Factory	186	—
Ours (full)	Factory	**226**	53.8
Greedy + heuristics	Crowd	188	43.7
Ours (first phase)	Crowd	260	—
Ours (full)	Crowd	**545**	123.5

Table 5.1: Main results table. See text for details.

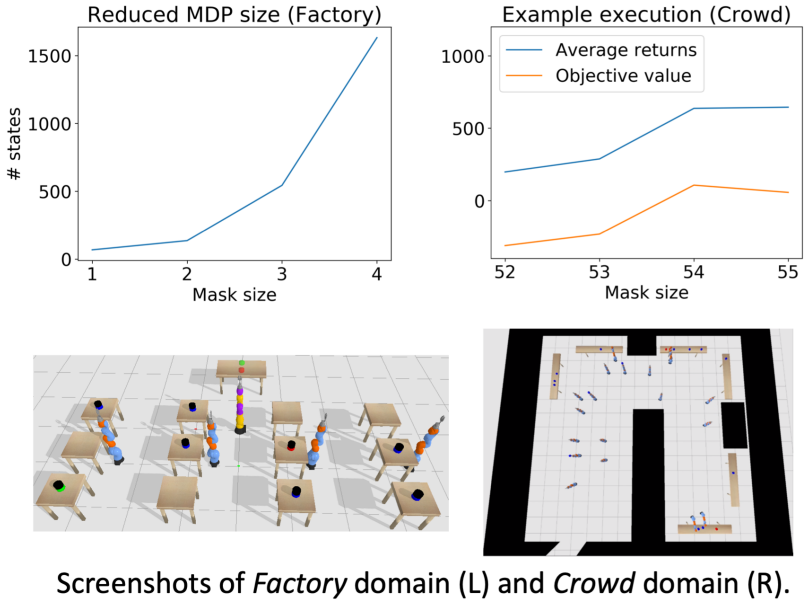


Figure 5-1: Additional experimental results (top row) and environment visualizations (bottom row). See text for details.

in the full MDP and searching for $\tilde{x}^*$ by brute force are prohibitively expensive. All results use $n_{\text{rollouts}} = 500$, $\tau_{\text{correl}} = 10^{-5}$, $\tau_{\text{variance}} = 0$, $n_1 = 250$, and $n_2 = 5$. We compare our algorithm with the *Greedy* one described earlier, which disregards the structure of the MDP. We also compare to only running our algorithm's first phase, which chooses the mask to be the estimated set of variables directly influencing the reward. The results are shown in Table 5.1 and Figure 5-1.

Discussion. In the *Factory* domain, the baseline greedy algorithm did not succeed even once at the task. The reason is that in this domain, several exogenous variables directly influence the reward function, but the greedy algorithm starts with an empty mask and only adds one variable at a time, and so cannot detect the improvement arising from adding in several variables at once. To give the baseline a fair

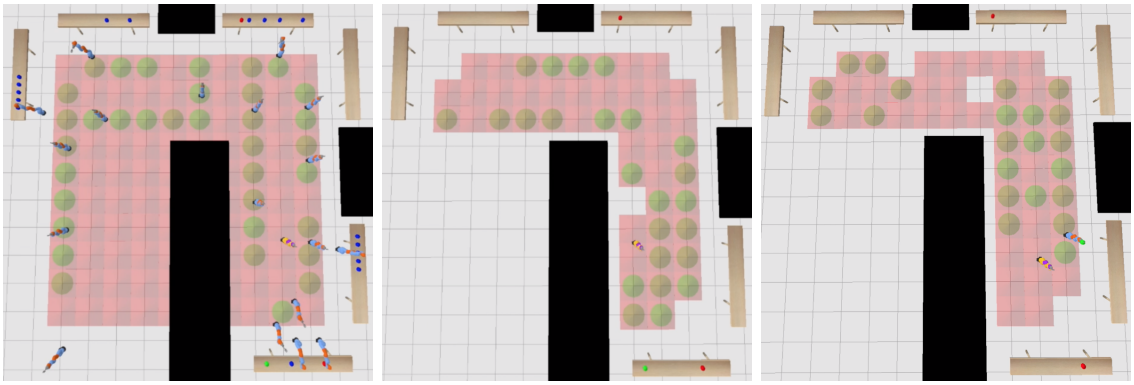


Figure 5-2: *Left*: Example of the full world. We control the singular gold-and-purple robot in the environment; the others follow fixed policies and are exogenous. *Medium, Right*: Examples of reduced models learned by the robot for Goals (1) and (2). The red squares on the ground describe which locations, within a discretization of the environment, the robot has learned to consider within its masked occupancy grid, while green circles denote currently occupied locations. Because all goals require manipulating objects on the tables, the robot recognizes that it does not need to consider occupancies in the lower-left quarter of the environment. For Goal (1), in which the other agents cannot manipulate the objects, the robot recognizes that it does not need to consider the states of any other agents. For Goal (2), the robot considers one of the other agents (here, holding the green object) within its reduced model, since this helps it better predict the dynamics of the objects.

chance, we initialized it by hand to a better mask for the *Crowd* domain, which is why it sees success there (*Greedy + heuristics* in the table). The results suggest that our algorithm, explicitly framed around *all* conditions of Theorem 1, performs well. The graph of the reduced MDP size shows the number of states in the *Factory* domain as a function of the number of exogenous variables included in the mask. The graph of the example execution in the *Crowd* domain shows that adding a 55th variable to the mask yields a decrease in the estimated objective even though the average returns slightly increase, due to the regularizer $|\tilde{x}|$.

Qualitatively, do the learned masks properly reflect different goals given to the robot?

An important characteristic of our algorithm is its ability to learn different masks based on what the goal is; we illustrate this concept with an example. Let us explore the masks resulting from two different goals in the *Crowd* domain: Goal (1) is for the robot to navigate to an object that cannot be moved by any of the other agents, and

Goal (2) is for the robot to navigate to an object that is manipulable by the other agents. In either case, the variables that directly affect the reward function are the object placements on the tables (which tell the robot where it must navigate to) and the occupancy grid (which helps the robot avoid crashing). However, for Goal (2), there is another variable that is important to consider: the states of any other agents that can manipulate the objects. This desideratum gets captured by the second phase of our algorithm: reasoning about the states of the other agents will allow the robot to better predict the dynamics of the object placements, enabling it to succeed at its task more efficiently and earn higher rewards. See Figure 5-2 for a visualization of this concept in our experimental domain, simulated using PyBullet [37].

Discussion. In a real-world setting, all of the exogenous variables could potentially be relevant to solving *some* problem, but typically only a small subset will be relevant to a *particular* problem. Under this lens, our method gives a way of deriving option policies in lower-dimensional subspaces.

What are the limitations of our approach?

Our experimentation revealed some limitations of our approach that are valuable to discuss. If the domains of the exogenous variables are large (or continuous), then it is expensive to compute the necessary mutual information quantities. To remedy this issue, one could turn to techniques for estimating mutual information, such as MINE [18]. Another limitation is that the algorithm as presented greedily adds one variable to the mask at a time, after the initial mask is built. In some settings, it can be useful to instead search over groups of variables to add in all at once, since these may contain information for better predicting dynamics that is not present in any single variable. Nevertheless, our experimentation has shown that our algorithm for mask-learning, in which an agent must choose a subset of the exogenous state variables of an MDP to reason about, leads to efficient planning in complicated domains.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Planning with Learned Object Importance in Large Problems

6.1 Motivation

A key aim in planning is to scale to large, real-world applications. In the previous chapters, we have discussed methods for projective abstraction via selecting constraints to impose on your own behavior and detecting irrelevant exogenous processes. However, these approaches may not scale effectively to very large problem instances, in particular applications that involve many *objects*, only some of which are important for any particular goal. For example, a household robot's internal state must include all objects relevant to *any* of its functions, but once it receives a *specific* goal, such as boiling potatoes, it should restrict its attention to only a small object set, such as the potatoes, pots, and forks, ignoring the hundreds or even thousands of other objects. If its goal were instead to clean the sink, the set of objects to consider would vary drastically.

In this chapter, we study projective abstraction for planning problems with large (but finite) universes of objects, where only a small subset need to be considered for any particular goal. Popular heuristic search planners [71, 66, 99] scale poorly in this regime (Figure 6-1), as they ground actions over the objects during preprocessing. Lifted planners [129, 36] avoid explicit grounding, but struggle during search; we

find that a state-of-the-art lifted planner [36] fails to solve any test problem in our experiments within the timeout (but it can usually solve the much smaller training problems). In this many-object setting, one would instead like to identify a small sufficient set of objects *before* planning, but finding this set is nontrivial and highly problem-dependent.

Our approach is to *learn to predict* subsets of objects that are sufficient for solving planning problems. This requires reasoning about discrete and continuous properties of the objects and their relations; moreover, generalizing to problems with more objects requires learning lifted models that are agnostic to object identity and count. We therefore propose a convolutional graph neural network architecture [132, 85, 15] that learns from a modest number (< 50) of small training problems and generalizes to hard test problems with many more objects. On the test problems, we use the network to predict a sufficient object set, remove all facts from the initial state and goal referencing excluded objects, and call an off-the-shelf planner on this reduced planning problem. For completeness, we wrap this procedure in an incremental loop that considers more objects until a solution is found.

Object importance prediction offers several advantages over alternative learning-based approaches: (1) it can treat the planner and transition model as black boxes; (2) its runtime does not depend on the number of ground actions (for a constant number of objects); (3) it permits efficient inference, therefore contributing negligibly to the overall planning time; and (4) it allows for a large margin of error in one direction, since the planning time can improve substantially even if only some irrelevant objects are excluded (Figure 6-1).

Gathering training data can be challenging in this setting because it requires labels of which objects are relevant; it would be impractical to assume that such labels are given. Instead, we propose a greedy approximate procedure for generating these labels automatically, which is only conducted in the relatively small training problems.

In experiments, we consider classical planning, probabilistic planning, and robotic task and motion planning, with test problems containing hundreds or thousands of objects. Our method, **P**lanning with **L**earned **O**bject **I**mportance (PLOI), results in

planning that is much more efficient than several baselines, including policy learning [60, 131] and partial action grounding [56]. We conclude that object importance prediction is a simple, powerful, and general mechanism for planning in large instances with many objects.

6.2 Related Work

Planning with Many Objects. Planning for problem instances that contain many objects is one of the main motivations for ongoing research in *lifted planning* [129, 36]. In STRIPS-like domains, lifted planners avoid the expensive preprocessing step of grounding the actions over all objects. Another way to alleviate the burden of grounding is to simplify the planning problem by creating *abstractions* [42, 44, 1]. Our object importance predictor can also be viewed as a type of learned abstraction selection [91, 130, 65].

Relational Representations for Learning to Plan. Our work uses graph neural networks (GNNs) [132, 85, 15], an increasingly popular choice for relational machine learning with applications to planning [151, 102, 133, 131]. One advantage of GNNs over logical representations [112, 97, 48] is that GNNs natively support continuous object-level and relational properties. We make use of this flexibility in our experiments, showing results in a simulated robotic environment.

Generalized Planning. Our work may be seen as an instance of *generalized planning*, which broadly encompasses methods for collectively solving a set of planning problems, rather than a single problem in isolation [76]. Other approaches to generalized planning include generalized policy learning [50, 60, 57], incremental search [86, 122], and heuristic or value function learning [7, 134, 133]. Incremental search and heuristic learning are complementary to our work and could be easily combined; generalized policy learning suggests a different mode of execution (executing the policy without planning) and we therefore include it as a baseline in our experiments.

The work perhaps most similar to ours is that of Gnad et al. [56], who propose

partial action grounding as another approach to generalized planning in large problems. Rather than predicting the probability that *objects* will be included in a plan (as we do), their approach predicts the probability that *ground actions* will be included. We include two versions of this approach as baselines in our experiments, including the implementation provided by the authors.

6.3 Problem Setting

We study relational planning problems (Section 2.2) with extraneous objects: ones that, if ignored, would make planning easier. The methods we propose are biased toward this subclass of planning problems and would not offer benefits in problems for which planning is easier, or only feasible, with all objects.

We consider the usual learning setting where we are first given a set of *training problems*, and then a separate set of *test problems*. All problems share $\mathcal{P}$, $\mathcal{A}$, and T, but may have different $\mathcal{O}$, I, and G. In general, the test problems will have a much larger set of objects $\mathcal{O}$ than the training problems; both training and test problems will have extraneous objects in $\mathcal{O}$. For simplicity, we assume properties have arity at most 2; higher-order ones can often be converted to an equivalent set of binary (arity 2) properties [131]. We treat object types as unary (arity 1) properties.

We are also given a black-box *planner*, denoted PLAN, which given a relational planning problem Π, produces either a plan or a policy, based on whether T is deterministic or stochastic. Going forward, we will not continue to make this distinction between plans and policies; in either case, at an intuitive level, PLAN produces ground actions that drive the agent toward its goal.

Our objective is to maximize the number of test problems solved within some time budget. Because the test problems contain many objects, and planners are often highly sensitive to this number, we will follow the broad approach of learning a model (on the training problems) that speeds up planning (on the test problems).

PLANNING WITH LEARNED OBJECT IMPORTANCE

Input: Planning problem $\Pi = \langle \mathcal{P}, \mathcal{A}, T, \mathcal{O}, I, G \rangle$. // See Section 6.3

Input: Object scorer f. // See Section 6.4

Hyperparameter: Geometric threshold γ.

// Step 1: compute importance scores

Compute $\text{score}(o) = f(o, I, G) \quad \forall o \in \mathcal{O}$

// Step 2: incremental planning

for $N = 1, 2, 3, \ldots$ **do**

  // Select objects above threshold

  $\hat{\mathcal{O}} \leftarrow \{o : o \in \mathcal{O}, \text{score}(o) \geq \gamma^N\}$

  // Create reduced problem & plan

  $\hat{\Pi} \leftarrow \text{REDUCEPROBLEM}(\Pi, \hat{\mathcal{O}})$

  $\pi \leftarrow \text{PLAN}(\hat{\Pi})$

  // Validate on original problem

  if $\text{ISSOLUTION}(\pi, \Pi)$ *or* $\hat{\mathcal{O}} = \mathcal{O}$ **then**

    return π

Algorithm 6: Pseudocode for PLOI. In practice, we perform two optimizations: (1) plan only when the object set $\hat{\mathcal{O}}$ changes, so that PLAN is called at most $|\mathcal{O}|$ times; and (2) assign a score of 1 to all objects named in the goal. See Section 6.4 for details and Figure 6-2 for an example.

6.4 Planning with Object Importance

In this section, we describe our approach for learning to plan efficiently in large problems. Our main idea is to learn a model that predicts a sufficient subset of the full object set. At test time, we use the learned model to construct a *reduction* of the planning problem, plan in the reduction, and validate the resulting plan in the original problem. To guarantee completeness, we repeat this procedure, incrementally growing the subset until a solution is found. This overall method — **P**lanning with **L**earned **O**bject **I**mportance (PLOI) — is summarized in Algorithm 6 and Figure 6-2.

We now describe PLOI in more detail, beginning with a more formal description of the reduced planning problem.

Definition 11 (Object set reduction)**.** *Given a planning problem* $\Pi = \langle \mathcal{P}, \mathcal{A}, T, \mathcal{O}, I, G \rangle$ *and a subset of objects* $\hat{\mathcal{O}} \subseteq \mathcal{O}$*, the* problem reduction $\hat{\Pi} = \text{REDUCEPROBLEM}(\Pi, \hat{\mathcal{O}})$ *is given by* $\hat{\Pi} = \langle \mathcal{P}, \mathcal{A}, T, \hat{\mathcal{O}}, \hat{I}, \hat{G} \rangle$*, where* $\hat{I}$ *(resp.* $\hat{G}$*) is* I *(resp.* G*) but with only properties over* $\hat{\mathcal{O}}$*.*

Intuitively, an object set reduction abstracts away all aspects of the initial state and goal pertaining to the excluded objects, and disallows any ground actions that involve these objects. This can result in a dramatically simplified planning problem, but may also result in an oversimplification to the point where planning in the reduction results in an invalid solution, or no solution at all. To distinguish such sets from the useful ones we seek, we use the following definition.

Definition 12 (Sufficient object set)**.** *Given a planning problem* $\Pi = \langle \mathcal{P}, \mathcal{A}, T, \mathcal{O}, I, G \rangle$ *and planner* PLAN*, a subset of objects* $\hat{\mathcal{O}} \subseteq \mathcal{O}$ *is* sufficient *if* $\pi = \textsc{Plan}(\hat{\Pi})$ *is a solution to* Π*, where* $\hat{\Pi} = \textsc{ReduceProblem}(\Pi, \hat{\mathcal{O}})$.

In words, an object set is *sufficient* if planning in the corresponding reduction results in a valid solution for the original problem. An object set that omits crucial objects, like a key needed to unlock a door or an obstacle that must be avoided, will not be sufficient: planning will fail without the key, and validation will fail without the obstacle. Trivially, the complete set of objects $\mathcal{O}$ is always sufficient if the planning problem is satisfiable and the planner is complete. However, we would like to identify a *small* sufficient set that permits faster planning. We therefore aim to learn a model that predicts such a set for a given initial state and goal.

6.4.1 Scoring Object Importance Individually

We wish to learn a model that allows us to identify a small sufficient subset of objects given an initial state, goal, and complete set of objects. There are three basic requirements for such a model. First, since our ultimate objective is to improve planning time, the model should be fast to query. Second, since we want to optimize the model from a modest number of training problems, the model should permit data-efficient learning. Finally, since we want to maintain completeness when the original planner is complete, the model should allow for some recourse when the first subset it predicts does not result in a valid solution.

These requirements preclude models that directly output a subset; such models offer no obvious recourse when the predicted subset turns out to be insufficient. More-

over, models that reason about sets of objects are, in general, likely to require vast amounts of training data and may require exorbitant time during inference.

We instead choose to learn a model $f : \mathcal{O} \times \mathcal{S} \times \mathcal{G} \to (0, 1]$ that scores objects *individually*. The output of the model $f(o, I, G)$ can be interpreted as the probability that the object o will be included in a small sufficient set for the planning problem $\langle \mathcal{P}, \mathcal{A}, T, \mathcal{O}, I, G \rangle$. We refer to this output score as the *importance* of an object. To get a candidate sufficient subset $\hat{\mathcal{O}}$ from such a model, we can simply take all objects with importance score above a threshold $0 < \gamma < 1$.

For the graph neural network architecture we will present in Section 6.5, this inference is highly efficient, requiring only a single inference pass. This parameterization also affords efficient learning, since as discussed at the end of this section, the loss function decomposes as a sum over objects. As an optimization, we always include in $\hat{\mathcal{O}}$ all objects named in the goal, since such objects must be in any sufficient set.

Another immediate advantage of predicting scores for objects individually is that there is natural recourse when the first candidate set $\hat{\mathcal{O}}$ does not succeed: simply lower the threshold γ and retry. In practice, we lower the threshold geometrically (see Algorithm 6), guaranteeing completeness.

Lemma 1 (PLOI is complete)**.** *Given any object scorer $f : \mathcal{O} \times \mathcal{S} \times \mathcal{G} \to (0, 1]$, if the planner* PLAN *is complete, then Algorithm 6 is complete.*

Proof. Since the codomain of f excludes 0, there exists an $\epsilon > 0$ s.t. $\{o : o \in \mathcal{O}, f(o, I, G) \geq \epsilon\} = \mathcal{O}$. Furthermore, $0 < \gamma < 1$, so there exists an iteration N s.t. $\gamma^N < \epsilon$. Therefore, even in the worst case, we will eventually run PLAN on the original problem and return the result. □

In predicting scores for objects individually, we have made the set prediction problem tractable by restricting the hypothesis class, but it is important to note that this restriction makes it impossible to predict certain object subsets. For example, in planning problems where a particular number of "copies" of the same object are required, e.g., three eggs in a recipe or five nails for assembly, individual object scoring

can only predict the same score for all copies. In practice, we find that this limitation is sharply outweighed by the benefits of efficient learning and inference.

In Section 6.5, we will present a graph neural network architecture for the object scorer f that is well-suited for relational domains. Before that, however, we describe a general methodology for learning f on the set of training problems.

6.4.2 Training with Supervised Learning

We now describe a general method for learning an object scorer f given a set of training problems $\mathbf{\Pi}_{\text{train}} = \{\Pi_1, \Pi_2, ..., \Pi_M\}$, where each $\Pi_i = \langle \mathcal{P}, \mathcal{A}, T, \mathcal{O}_i, I_i, G_i \rangle$. The main idea is to cast the problem as supervised learning. From each training problem Π_i, we want to extract input-output pairs $\{((o, I_i, G_i), y)\}$, where $o \in \mathcal{O}_i$ is each object from the full set for the problem, and $y \in \{0, 1\}$ is a binary label indicating whether o should be predicted for inclusion in the small sufficient set. The overall training dataset for supervised learning, then, will contain an input-output pair for every object, for each of the M training problems.

The y labels for the objects are *not* given, and moreover, it can be challenging to exactly compute a minimal sufficient object set, even in small problem instances. We propose a simple approximate method for automatically deriving the labels. Given a training problem Π_i, we perform a greedy search over object sets: starting with the full object set $\mathcal{O}_i$, we iteratively remove an individual object from the set, accepting the new set if it is sufficient, until no more individual objects can be removed without violating sufficiency. All objects in the final sufficient set are labeled with $y = 1$, while the remaining objects are labeled with $y = 0$. This procedure, which requires planning several times per problem instance with full or near-full object sets to check sufficiency, exploits the fact that the training problems are much smaller and easier than the test problems.

It should be noted that the aforementioned greedy procedure is an approximation, in the sense that there may be some smaller sufficient object set than the one returned. To illustrate this point, consider a domain with a certain number of widgets where the only parameterized action is `destroy(?widget)`. Suppose the goal is to be left

with a number of widgets that is divisible by 10, and that the full object set itself has 10 widgets. The greedy procedure will terminate after the first iteration, since no object can be removed while maintaining sufficiency. However, the empty set is actually sufficient because it induces the empty plan, which trivially satisfies this goal. Despite such possible cases, this greedy procedure for deriving the training data does well in practice to identify small sufficient object sets.

With a dataset for supervised learning in hand, we can proceed in the standard way by defining a loss function and optimizing model parameters. To permit data-efficient learning, we use a loss function that decomposes over objects:

$$\mathcal{L}(\mathbf{\Pi}_{\text{train}}) = \sum_{i=1}^{M} \sum_{o_j \in \mathcal{O}_i} \mathcal{L}_{\text{obj}}(y_{ij}, f(o_j, I_i, G_i)),$$

where y_{ij} is the binary label for the j^{th} object in the i^{th} training problem, and $f(o_j, I_i, G_i) \in (0, 1]$. We use a weighted binary cross-entropy loss for $\mathcal{L}_{\text{obj}}$, where the weight (10 in experiments) gives higher penalty to false negatives than false positives, to account for class imbalance.

6.5 Object Importance Scorers as GNNs

We have established individual object importance scorers $f : \mathcal{O} \times \mathcal{S} \times G \rightarrow (0, 1]$ as the model that we wish to learn. We now turn to a specific model class that affords gradient-based optimization, data-efficient learning, and generalization to test problems with new and many more objects than were seen during training. Graph neural networks (GNNs) offer a flexible and general framework for learning functions over graph-structured data [85]. GNNs employ a relational bias that is well-suited for our setting, where we want to make predictions based on the relations that objects are involved in, but we do not want to overfit to the particular identity or number of objects in the training problems [15]. Such a relational bias is crucial for generalizing from training with few objects to testing with many. Furthermore, GNNs can be used in domains with continuous properties, unlike traditional inductive

logic programming methods [112, 97]. We stress that other modeling choices are possible, such as statistical relational learning methods [89], as long as they are lifted, relational, efficiently learnable, and able to handle continuous properties; we have chosen GNNs here because they are convenient and well-supported.

The input to a GNN is a directed graph with nodes $\mathcal{V}$ and edges $\mathcal{E}$. Each node $v \in \mathcal{V}$ has a feature vector $\phi_{\text{node}}(v) \in \mathbb{R}^{D^{\text{in}}_{\text{node}}}$, where $D^{\text{in}}_{\text{node}}$ is the (common) dimensionality of these node feature vectors. Each edge $(v_1, v_2) \in \mathcal{E}$ has a feature vector $\phi_{\text{edge}}(v_1, v_2) \in \mathbb{R}^{D^{\text{in}}_{\text{edge}}}$, where $D^{\text{in}}_{\text{edge}}$ is the (common) dimensionality of these edge feature vectors. The output of a GNN is another graph with the same topology, but the node and edge features are of different dimensionalities: $D^{\text{out}}_{\text{node}}$ and $D^{\text{out}}_{\text{edge}}$ respectively. Internally, the GNN passes messages for K iterations from edges to sink nodes and from source nodes to edges, where the messages are determined by fully connected networks with weights shared across nodes and edges. We use the standard Graph Network block [15], but other choices are possible. Like other neural networks, GNNs can be trained with gradient descent.

We now describe how object importance scoring can be formulated as a GNN. The high-level idea is to associate each object with a node, each unary property (including object types) with an input node feature, each binary property with an input edge feature, and each importance score with an output node feature. See Figure 6-3 for an example.

Given a planning problem with object set $\mathcal{O}$, we construct input and output graphs where each node $v \in \mathcal{V}$ corresponds to an object $o \in O$. In the output graph, there is a single feature for each node; i.e., $D^{\text{out}}_{\text{node}} = 1$. This feature represents the importance score $f(o, I, G)$ of each object o. The edges are ignored in the output graph.

The input graph is an encoding of the initial state I and goal G. Recall that the initial state I is defined by an assignment of all ground properties ($\mathcal{P}$ over $\mathcal{O}$) to values, and that all properties are unary (arity 1) or binary (arity 2). Each unary property, which includes object types, corresponds to one dimension of the input node feature vector $\phi_{\text{node}}(o)$. Each binary property corresponds to *two* dimensions of the input edge feature vector $\phi_{\text{edge}}(o_1, o_2)$: one for each of the two orderings of the objects

(see Figure 6-3).

Recall that a goal G is characterized by an assignment of some subset of ground properties to values. Unlike the initial state, not all ground properties must appear in the goal; in practice, goals are typically very sparse relative to the state. For each ground property, we must indicate whether it appears in the goal, and if so, with what assignment. For each unary property, we add two dimensions to the input node feature vector $\phi_{\text{node}}(o)$: one indicating the presence (1) or absence (0) of the property, and the other indicating the value, with a default of 0 if the property is absent. Similarly, for each binary property, we add four dimensions to the input edge feature vector $\phi_{\text{edge}}(o_1, o_2)$: two for the orderings multiplied by two for presence and assignment.

For STRIPS-like domains where properties are predicates, we make two small simplifications. First, to make the graph computations more efficient, we sparsify the edges by removing any edge whose features are all zeros. Second, in the common case where goals do not involve negation, we note that the presence/absence dimension will be equivalent to the assignment dimension; we thus remove the redundant dimension. Figure 6-3 makes use of these simplifications.

Given a test problem and trained GNN, we construct an input graph, feed it to the GNN to get an output graph, and read off the predicted importance scores for all objects. This entire procedure needs only one inference pass (with $K = 3$ message passing iterations) to predict all object scores; it takes just a few milliseconds in our experiments.

6.6 Experiments

In this section, we present empirical results for PLOI and several baselines. We find that PLOI improves the speed of planning significantly over all these baselines.

	Pure Plan		PLOI (Ours)		Rand Score		Neighbors		Policy		ILP AG		GNN AG	
Domains	Time	Fail	Time	Fail	Time	Fail	Time	Fail	Time	Fail	Time	Fail	Time	Fail
Blocks	7.47	0.00	**0.62**	0.00	49.99	0.00	**0.52**	0.00	7.25	0.74	2.33	0.00	52.95	0.23
Logistics	**8.55**	0.00	**6.44**	0.00	42.05	0.00	**15.40**	0.00	–	1.00	–	1.00	49.31	0.81
Miconic	87.71	0.06	**21.64**	0.04	–	1.00	93.86	0.98	–	1.00	–	1.00	–	1.00
Ferry	**12.64**	0.00	**7.52**	0.00	43.79	0.10	39.66	0.00	34.78	0.91	33.77	0.00	–	1.00
Gripper	24.48	0.00	**0.47**	0.00	56.58	0.29	37.63	0.00	28.94	0.60	5.71	0.20	86.29	0.95
Hanoi	**3.19**	0.00	**3.39**	0.00	**3.47**	0.00	4.63	0.00	–	1.00	6.15	0.00	7.55	0.00
Exploding	11.52	0.30	**0.81**	0.30	44.96	0.32	1.08	0.29	10.18	0.89	4.69	0.19	48.53	0.38
Tireworld	24.58	0.01	**4.38**	0.08	44.09	0.29	47.13	0.00	30.36	0.10	–	1.00	63.03	0.66
PyBullet	–	1.00	**2.05**	0.00	–	1.00	8.58	0.01	–	–	–	–	–	–

Table 6.1: On test problems, failure rates within a 120-second timeout and planning times in seconds over successful runs. All numbers report a mean across 10 random seeds, which randomizes both GNN training (if applicable) and testing. All times are in seconds; bolded times are within two standard deviations of best. AG = action grounding. Policy and AG baselines are not run for PyBullet because these methods cannot handle continuous actions. Across all domains, PLOI is consistently best and usually at least two standard deviations better than all other methods.

6.6.1 Experimental Design

Baselines. We consider several baselines in our experiments, ranging from pure planning to state-of-the-art methods for learning to plan. All GNN baselines are trained with supervised learning using the set of plans found by an optimal planner on small training problems.

- **Pure planning.** Use the planner PLAN on the complete test problems, with all the objects.
- **Random object scoring.** Use the incremental procedure described in Section 6.4, but instead of using a trained GNN to score the importance of each object, give each object a uniformly random importance score between 0 and 1. This baseline can be understood as an ablation that removes the GNN from our system.
- **Neighbors.** This is a simple heuristic approach that incrementally tries planning with all objects that are connected by at most L steps in the graph of relations to any object named in the goal, for $L = 0, 1, 2, \ldots$. If a plan has not been found even after all objects connected to a goal object have been considered, we fall back to pure planning for completeness.
- **Reactive policy.** Inspired by other works that learn reactive, goal-conditioned policies for planning problems [60, 131], we modify our GNN architecture to predict a ground action per timestep. The input remains the same, but the output

has two heads: one predicts a probability over actions $\mathcal{A}$, and the other predicts, for every parameter of that action, a probability over objects. At test time, we compute all valid actions in each state and execute the one with the highest probability under the policy. This baseline does not use PLAN at test time.

- **ILP action grounding.** This baseline is the method presented by Gnad et al. [56], described in Section 6.2, with the best settings they report. We use the implementation provided by the authors for both training and test. We use the SVR model with round robin queue ordering, and incremental grounding with increment 500.
- **GNN action grounding.** We also investigate using a GNN in place of the inductive logic programming (ILP) model used by the previous baseline [56]. To implement this, we modify our GNN architecture to take as input a ground action in addition to the state and goal, and output the probability that this ground action should be considered when planning.

As mentioned in Section 6.1, we also attempted to compare to a state-of-the-art lifted planner [36], using the implementation provided by the authors. However, we found that this planner was unable to solve any of our test problems in any domain, although it was usually able to solve the (much smaller) training problems.

Domains. We evaluate on 9 domains: 6 classical planning, 2 probabilistic planning, and 1 simulated robotic task and motion planning. We chose several of the most standard classical and probabilistic domains from the International Planning Competition (IPC) [27], but we procedurally generated problems involving many more objects than is typical. In all domains, we train on 40 problem instances and test on 10 much larger ones. For interacting with IPC domains, we use the PDDLGym library [135], version 0.0.2.

- **Tower of Hanoi**. The classic Tower of Hanoi domain, in which disks must be moved among three pegs. All objects are always necessary to consider in this domain; we have included this domain to show that PLOI does not have much overhead on top of pure planning in this situation. We train on problems containing 4-9 disks and test on problems containing 10-15 disks. The plan lengths

for training (test) problems range from 1-63 (511-8191).

- **Blocks**. Problems involve blocks in small piles on a table, and the goal is to configure a particular small subset of the blocks into a tower. We train on problems containing 15-32 blocks and test on problems containing 100-150 blocks. Test goals involve 20-25 blocks. The plan lengths for training (test) problems range from 4-10 (26-64).
- **Gripper**. Problems involve one robot that can pick and place balls and move to different rooms. A goal is an assignment of a subset of the balls to rooms. We train on problems containing 36-52 objects and test on problems containing 100-200 objects. Test goals involve placing 10-20 balls in random rooms. The plan lengths for training (test) problems range from 7-19 (27-107).
- **Miconic**. Passengers in buildings with elevators are trying to reach particular floors. We train on problems involving 33-63 objects. We test on problems with 20-30 floors, 2 passengers per building, and 100 buildings, for a total of over 2000 objects. Goals involve moving one passenger per building to their desired floor. The plan lengths for training (test) problems range from 11-12 (894-917).
- **Ferry**. A ferry transports cars to various locations. We train on problems with 13-21 objects and test on problems with 250-350 objects. Goals involve moving 3 cars to random locations. The plan lengths for training (test) problems range from 7-12 (14-17).
- **Logistics**. Trucks and airplanes are used to transport crates to cities. We train on problems with 13-40 objects. Test problems have around 50 airplanes, 20 cities, 20 trucks, 20-50 locations, and 20 crates. Goals involve moving around 20 crates to random cities. The plan lengths for training (test) problems range from 5-32 (66-203).
- **Exploding Blocks**. A probabilistic IPC domain, where whenever the agent interacts with a block, there is a chance that the block or the table are irreversibly destroyed; no policy can succeed all the time in this domain. Problem sizes are the same as in Blocks.
- **Triangle Tireworld**. A probabilistic IPC domain, containing an agent that must

navigate through cities, and has a chance of getting a flat tire on each timestep. The agent can only change its tire at certain cities that have spare tires. It is always possible to reach the goal city by simply avoiding cities that do not have spare tires. We test on worlds with side length around 50.

- **PyBullet robotic simulation** [37]. In this domain with continuous object properties, a fixed robot arm mounted on the center of a table must interact with a particular can on the table while avoiding all other irrelevant cans. See Figure 6-4 for details and a visualization. To encode this domain in our GNN, we treat the continuous object poses as node features. Test problems have around 1000 irrelevant cans on the table. The goal always involves manipulating a single can.

Experimental Details. We use Fast Downward [66] in the `LAMA-first` mode as the base classical planner for test time in all experiments. To gather training data with an optimal planner, we use Fast Downward in `seq-opt-lmcut` mode. For planning in probabilistic domains, we use single-outcome determinization and replanning [154]. For TAMP in the PyBullet experiment, we use PDDLStream [53] in `focused` mode. All experiments were performed on Ubuntu 18.04 using four cores of an Intel Xeon Gold 6248 processor, with 10GB RAM per core. We use $\gamma = 0.9$ for all experiments. GNNs are implemented in PyTorch, version 1.5.0. All GNNs 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 for 1000 epochs. The batch size is 16. Preliminary experiments with ℓ_2 regularization, dropout, and hyperparameter search yielded no consistent improvements for any of the methods.

6.6.2 Main Results and Discussion

All experiments are conducted over 10 random seeds. Table 6.1 shows failure rates within a 120-second timeout and average planning time on successful runs. Initial experimentation found no significant difference in our results between 300-second and 120-second timeouts. Across all domains, PLOI consistently plans much faster

than all the other methods. In some domains, such as Gripper, PLOI is faster than **pure planning** by two orders of magnitude. In the case of Hanoi, where all objects are necessary, we see that PLOI is comparable to pure planning, which confirms the desirable property that PLOI reduces to pure planning with little overhead in problems where all objects are required.

Comparing PLOI with the **random object scoring** baseline, we see that PLOI performs much better in all domains other than Hanoi. This comparison suggests that the GNN is crucial for the efficient planning that PLOI attains. To further analyze the impact of the GNN, we plot the number of iterations (N in Algorithm 6) that are needed until the incremental planning loop finds a solution, for both PLOI and random object scoring (Figure 6-5). The dramatic difference between the two methods confirms that the GNN has learned a very meaningful bias, allowing a sufficient object set to be consistently found in less than 5 iterations, and often just 1.

The key difference between PLOI and the **action grounding (AG)** baselines is that PLOI predicts which *objects* would be sufficient for a planning problem, while the AG baselines predict which *ground actions* would be sufficient for a planning problem. Empirically, PLOI performs better than all the AG baselines, due to the fact that PLOI has comparatively little overhead, while the AG baselines spend significant time during inference on trying to score all the possible ground actions, of which there are significantly more than the number of objects. Another benefit of PLOI is that it uses PLAN as a black box, whereas the AG baselines must modify the internals of PLAN, e.g. by changing the set of ground actions instantiated during translation or followed during search.

The **neighbors** baseline performs well in some domains, but not in others; it performs particularly poorly in domains where the agent must consider an object that does not share a relation with some other important one, e.g. a ferry in the Ferry domain. Looking now at the **policy** baseline, we see that it is generally quite slow. This is because even though the policy baseline does not use PLAN, it takes time to compute all valid actions and query the policy GNN to find the most probable one on every timestep; by contrast, PLOI only performs inference once, on the first timestep.

Moreover, the policy has a high failure rate relative to the planning baselines, since there is no recourse when it does not succeed.

Finally, the results in the continuous PyBullet domain suggest that PLOI is able to yield meaningful improvements over an off-the-shelf task and motion planning system. Learning in the hybrid state and action spaces of task and motion planning domains is extremely challenging in general; reactive policy learning is typically unable to make meaningful headway in these domains. Moreover, it is not possible to apply the action grounding approach due to the infinite number of ground actions (e.g., poses for grasping a can). PLOI works well here because it uses a planner in conjunction with making predictions at the level of the (discrete) object set, not the (continuous) ground action space.

6.6.3 Additional Results

Here we report additional experiments and results.

6.6.3.1 Effect of Message Passing Iterations (K)

We used $K = 3$ message passing iterations for all graph neural networks. To better understand the impact of this hyperparameter on our main results, we reran PLOI on Blocks, varying K from 1 to 5. As seen in Figure 6-6, results are robust for $2 \leq K \leq 5$, but performance drops off heavily for $K = 1$, suggesting that some propagation through the GNN matters. In other domains, we would similarly expect $K = 1$ to be insufficient, but we may not always expect $K = 2$ to suffice. Generally, setting K appropriately involves a trade-off: too low values may prevent the model from fitting the data, while too large values may slow computation and risk overfitting. A hyperparameter search increasing from $K = 1$ should do well to identify an appropriate value for any domain.

6.6.3.2 Effect of Number of Training Problems

We used < 50 training problems in all domains, with 40 used in Blocks. To better understand the impact of the number of training problems on our main results, we reran PLOI on Blocks, varying the number of training problems between 2 and 40. As seen in Figure 6-7, performance peaks very quickly, starting at 3 and remaining robust for > 3. We would not necessarily expect so few examples to suffice for the other domains.

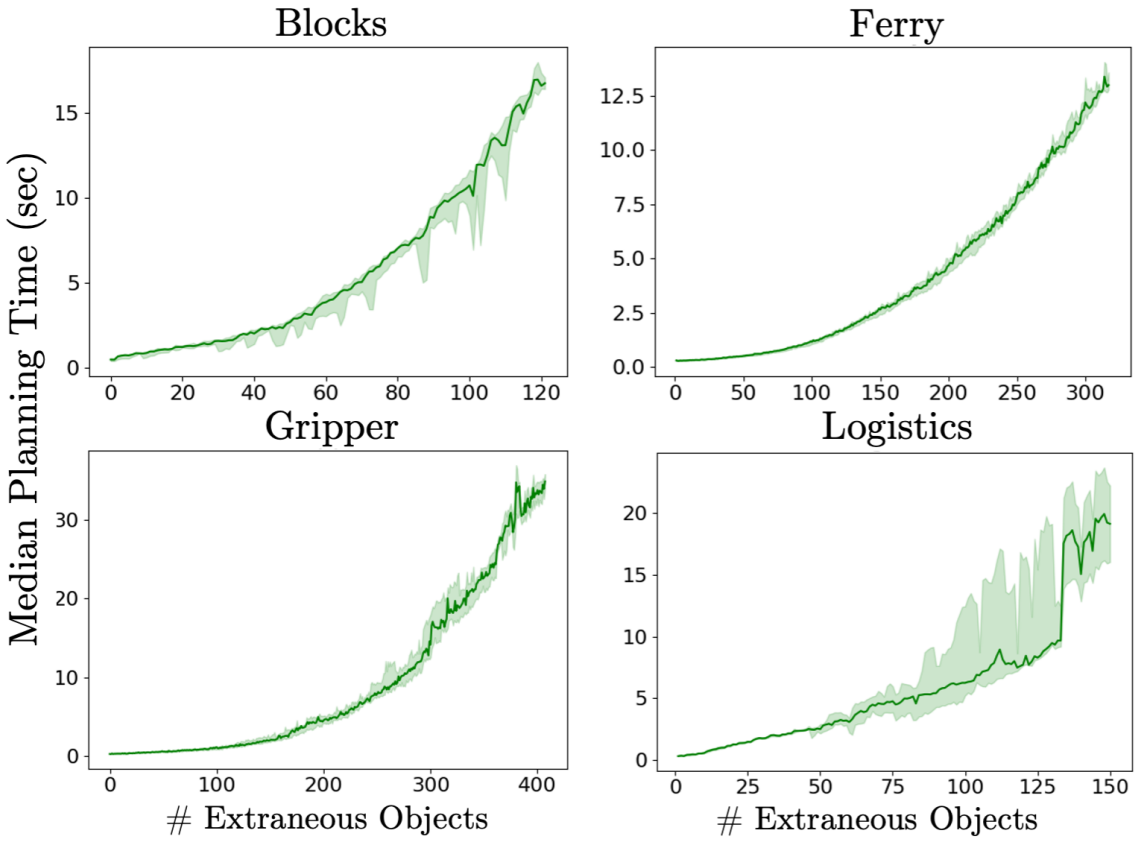


Figure 6-1: Time taken by Fast Downward [66] in the `LAMA-first` mode on various IPC domains, as a function of the number of extraneous objects in the problem. The x-axis is the number of objects added to a small sufficient set (Definition 12). Curves show a median across 10 problems; shaded regions show the 25th to 75th percentiles. We can see that planning time gets substantially worse as the number of extraneous objects increases; real-world applications of planning will often contain large numbers of such objects for a particular goal. In this chapter, we learn to predict a small subset of objects that is sufficient for planning, leading to significantly faster planning than both Fast Downward on its own and other learning-based grounding methods.

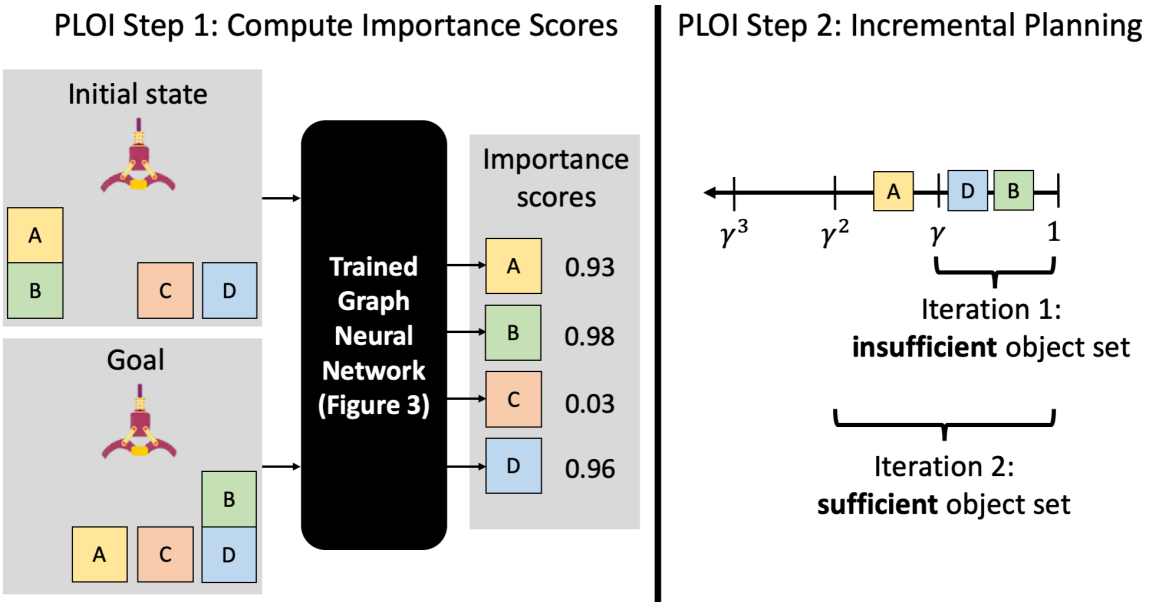


Figure 6-2: Overview of our method, PLOI, with an example. *Left:* To solve this problem, the robot must move block A to the free space, then stack B onto D. The GNN computes the per-object importance score. Block C is irrelevant, and therefore it receives a low score of 0.03. *Right:* We perform incremental planning. In this example, $\gamma = 0.95$, so that $\gamma^2 \approx 0.9$. The first iteration tries planning with the object set {B, D}, which fails because it does not consider the obstructing A on top of B. The second iteration succeeds, because the object set {A, B, D} is sufficient for this problem.

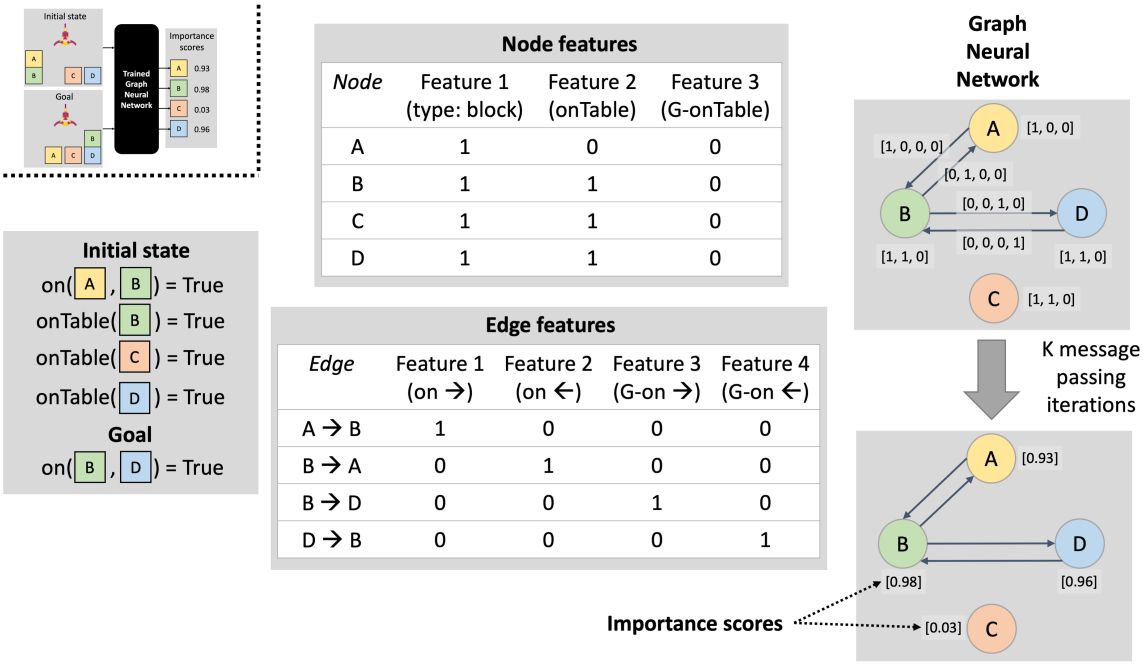


Figure 6-3: Illustration of object importance scoring with GNNs. We consider the same planning problem example as in Figure 6-2. A node is created for each of the four objects, with features determined by the unary properties in the initial state and goal. An edge is created for each ordered pair of objects, with features determined by the binary properties, and with trivial edges excluded. These nodes and edges constitute the input graph to a GNN, which performs $K = 3$ message passing iterations before outputting another graph with the same topology. Each output node is associated with the object's importance score.

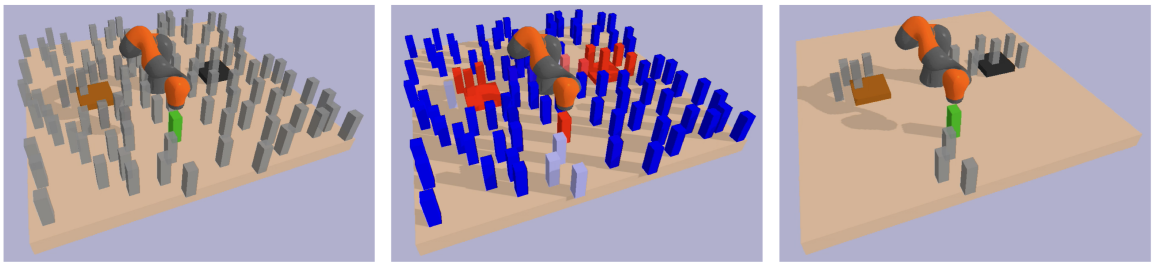


Figure 6-4: Example problem from the PyBullet domain. *Left:* The robot arm must move the target can (green) to the stove (black) and then to the sink (brown) while avoiding other cans (gray). *Middle:* GNN importance scores for this problem, scaled from blue (low importance) to red (high importance). We can see that cans surrounding the sink, stove, and target have been assigned higher importance score, meaning the GNN has reasoned about geometry. *Right:* The reduced problem in which the robot plans. Only objects with importance score above some threshold remain in the scene.

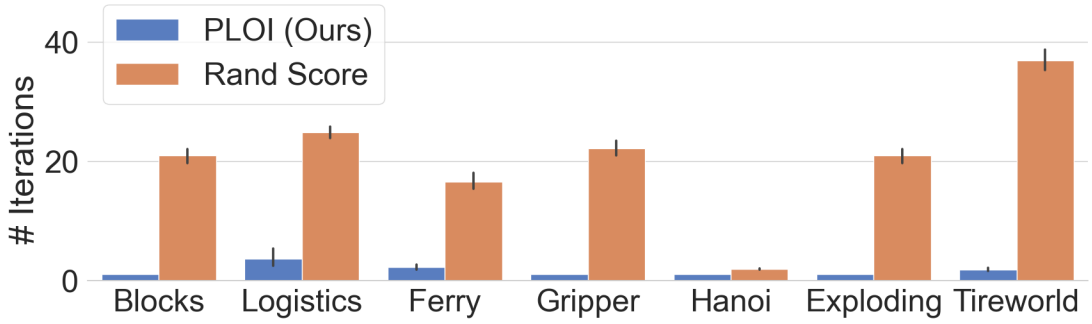


Figure 6-5: Number of iterations (N in Algorithm 6) needed until incremental planning finds a solution, for both PLOI and random object scoring. Results are averaged over 10 seeds, with standard deviations shown as vertical lines. Miconic and PyBullet are not included because random object scoring never succeeded in this domain. We can see that the GNN has learned a meaningful bias, allowing a sufficient object set to be consistently found in fewer than 5 iterations.

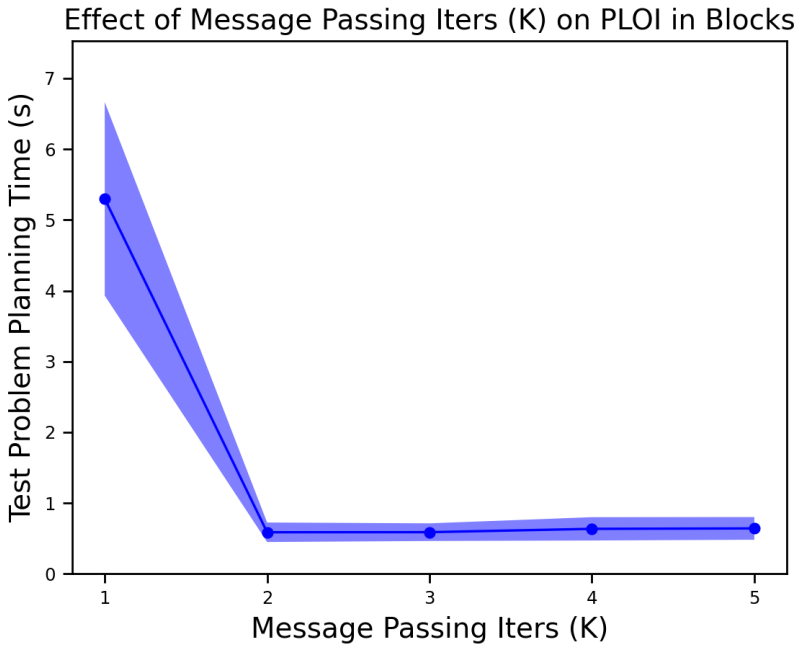


Figure 6-6: Effect of message passing iterations (K) on the performance of PLOI in Blocks. Results are averaged over 10 seeds, with standard deviations shown as shaded areas.

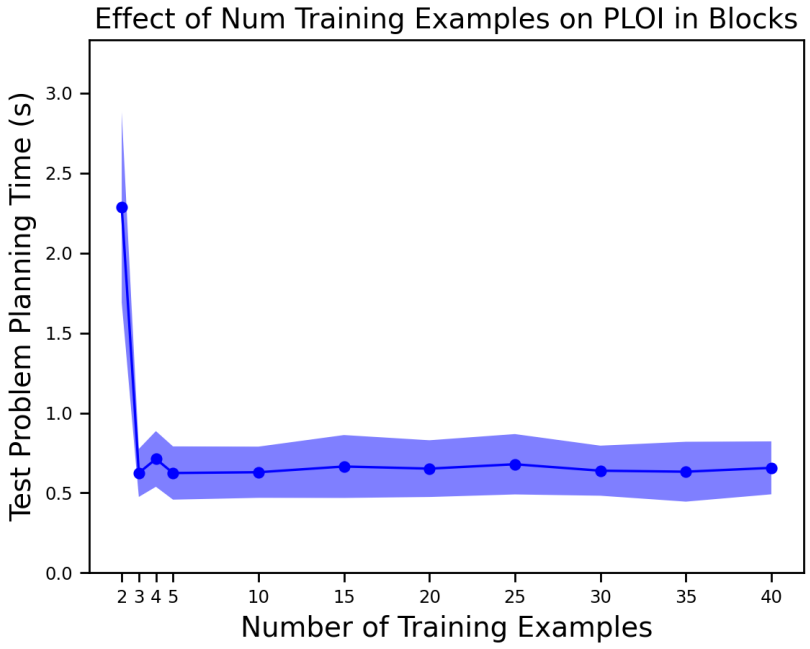


Figure 6-7: Effect of number of training examples on the performance of PLOI in Blocks. Results are averaged over 10 seeds, with standard deviations shown as shaded areas.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Conclusion and Future Work

In this thesis, we described novel frameworks for learning state and action abstractions that are optimized for effective and efficient planning. Throughout the chapters, we showed how to learn symbolic abstractions for bilevel planning (Chapter 3), presented a method for learning to generate context-specific abstractions of MDPs (Chapter 4), formalized and gave a tractable algorithm for the exogenous variable mask-learning problem (Chapter 5), and introduced a simple yet powerful and general mechanism for planning in large problem instances containing many objects (Chapter 6).

We explored the idea of learning *abstractions* as a general mechanism for ameliorating the intractability of planning. We studied two forms of task-specific abstractions, with a common goal of *using an abstraction for effective and efficient planning.* The first form was *neuro-symbolic, relational* abstractions, which allow a robot to plan to long horizons in continuous spaces by learning STRIPS-style predicates and operators and neural network samplers, then using them for bilevel hierarchical planning. The second form was *projective* abstractions, which detect and drop irrelevant variables from a factored planning problem to make it easier to solve.

In Chapter 3, experiments across four robotic planning environments showed that our framework learns relational, neuro-symbolic abstractions that generalize over object identities, can efficiently solve long-horizon held-out tasks, and are even able to outperform manually specified abstractions. Key areas for future work include (1)

relaxing the assumption that controllers are provided; (2) learning better abstractions from even fewer demonstrations by performing active learning to gather more data online; (3) expanding the expressivity of the grammar to learn more sophisticated predicates; and (4) applying the ideas presented in the chapter to non-deterministic and/or partially observed planning problems. For (1), the literature on option learning [138, 144] provides a starting point; in our setting, it would be necessary to not only learn the initiation sets, policies, and termination conditions of the options [143], but also figure out how to *segment* the demonstrations appropriately in the first place to learn these options, since now these demonstrations would be just state sequences. For (2), we hope to investigate how relational exploration algorithms [95, 32] might be useful as a mechanism for an agent to decide what actions to execute, toward the goal of building better state and action abstractions. For (3), we can take inspiration from program synthesis, especially methods that can learn programs with continuous parameters [49]. Finally, for (4) we would like to draw insights from recent advances in task and motion planning, in both the stochastic setting [64] and the partially observed setting [54].

In Chapter 4, we showed that CAMPs achieve more efficient planning while retaining high rewards. There are several clear directions for future work. On the learning side, one interesting question is whether factorizations of initially unfactored MDPs can be automatically discovered in a way that leads to useful CAMPs. Another direction to pursue is learning the task featurizer ϕ, which we assumed to be given in our problem formulation. Following [82], it could also be useful to extend the methods we have presented here so that multiple contexts can be imposed in succession at test time, using the performance of previous contexts to inform the choice of future ones. However, note that such a method would lead to an increase in computational cost, possibly to the detriment of the overall objective we formulated.

In Chapter 5, we developed an efficient algorithm, based on approximate mutual information, for deciding which exogenous variables to keep in a planning problem. An important avenue for future work is to remove the assumption that the agent knows the partition of endogenous versus exogenous aspects of the state. An inter-

esting fact to ponder is that the agent can actually *control* this partition by choosing its actions appropriately. Thus, the agent can commit to a particular choice of exogenous variables in the world, and plan under the constraint of never influencing these variables. Another avenue for future work is to develop an incremental, real-time version of the algorithm, necessary in settings where the agent's task constantly changes.

In Chapter 6, we showed empirically that PLOI performs well across classical planning, probabilistic planning, and robotic task and motion planning. As PLOI makes use of a neural learner to inform black-box symbolic planners, we view it as a step toward the greater goal of integrated neuro-symbolic artificial intelligence [105, 118]. An immediate direction for future work would be to investigate the empirical impact of using a GNN as the importance scorer, versus techniques in statistical relational learning [89, 125]. Another direction would be to study how to apply PLOI to open domains, where the agent does not know in advance the set of objects that are in a problem instance. Addressing this kind of future direction can help learning-to-plan techniques like PLOI fully realize their overarching aim of solving large-scale, real-world planning problems.

Overall, the results presented in this thesis demonstrate that we can and should design autonomous agents that optimize a tradeoff between two competing criteria: *effectiveness* and *efficiency*. By learning abstractions with particularly defined objective functions that explicitly consider both criteria, it is possible to control where the capabilities of our autonomous systems lie on this spectrum. Broadly, this corresponds to a tradeoff between the time spent making decisions and the quality of those decisions. Looking to the future, I'd like to offer the following quote from a blog post I co-authored: "One day we would like to have robots that live in our houses and do all of the chores that we would rather not do: laundry, cooking, installing and fixing appliances, taking out the trash, and so on. When someone gives one of these robots a task, like 'make me pasta!', they'll want the robot to perform the task effectively and efficiently. They would be unhappy to receive a bowl full of trail mix or toothpaste or nothing at all; that would be ineffective. They would be similarly

disappointed to watch the robot remain motionless for hours as it ponders how to complete the task before it, as though it were asked to prove P != NP; that would be inefficient." Ultimately, we will need agents to be able to learn abstractions that are conducive to effective and efficient planning. I hope that as a field, we can keep these important considerations in mind as the literature on planning and learning continues to mature in the coming years and decades.

Bibliography

[1] David Abel, David Hershkowitz, and Michael Littman. Near optimal behavior via approximate state abstraction. In *International Conference on Machine Learning*, pages 2915–2923, 2016.

[2] David Abel, Nate Umbanhowar, Khimya Khetarpal, Dilip Arumugam, Doina Precup, and Michael Littman. Value preserving state-action abstractions. In *International Conference on Artificial Intelligence and Statistics*, pages 1639–1650, 2020.

[3] 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.

[4] Yusra Alkhazraji, Matthias Frorath, Markus Grützner, Malte Helmert, Thomas Liebetraut, Robert Mattmüller, Manuela Ortlieb, Jendrik Seipp, Tobias Springenberg, Philip Stahl, and Jan Wülfing. Pyperplan. `https://doi.org/10.5281/zenodo.3700819`, 2020.

[5] Barrett Ames, Allison Thackston, and George Konidaris. Learning symbolic representations for planning with parameterized skills. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 526–533, 2018.

[6] David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. In *Eighteenth national conference on Artificial intelligence*, pages 119–125, 2002.

[7] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

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

[9] Masataro Asai. Unsupervised grounding of plannable first-order logic representation from images. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 583–591, 2019.

[10] 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.

[11] Masataro Asai and Christian Muise. Learning neural-symbolic descriptive planning models via cube-space priors: The voyage home (to STRIPS). *arXiv preprint arXiv:2004.12850*, 2020.

[12] Fahiem Bacchus. AIPS 2000 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems. *AI magazine*, 22(3):47–47, 2001.

[13] Aijun Bai, Siddharth Srivastava, and Stuart J Russell. Markovian state and action abstractions for MDPs via hierarchical MCTS. In *International Joint Conference on Artificial Intelligence*, pages 3029–3039, 2016.

[14] Chris L Baker, Rebecca Saxe, and Joshua B Tenenbaum. Action understanding as inverse planning. *Cognition*, 113(3):329–349, 2009.

[15] 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.

[16] Jiri Baum, Ann E Nicholson, and Trevor I Dix. Proximity-based non-uniform abstractions for approximate planning. *Journal of Artificial Intelligence Research*, 43:477–522, 2012.

[17] James C Bean, John R Birge, and Robert L Smith. Aggregation in dynamic programming. *Operations Research*, 35(2):215–220, 1987.

[18] Mohamed Ishmael Belghazi, Aristide Baratin, Sai Rajeswar, Sherjil Ozair, Yoshua Bengio, Aaron Courville, and R Devon Hjelm. MINE: mutual information neural estimation. *arXiv preprint arXiv:1801.04062*, 2018.

[19] Pascal Bercher, Ron Alford, and Daniel Höller. A survey on hierarchical planning-one abstract idea, many concrete realizations. In *International Joint Conference on Artificial Intelligence*, pages 6267–6275, 2019.

[20] Dimitri P Bertsekas, David A Castanon, et al. Adaptive aggregation methods for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, 1988.

[21] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[22] 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.

[23] Craig Boutilier. Correlated action effects in decision theoretic regression. In *UAI*, pages 30–37, 1997.

[24] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[25] Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth conference on Uncertainty in artificial intelligence*, 1996.

[26] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[27] Daniel Bryce and Olivier Buffet. International planning competition uncertainty part: Benchmarks and results. In *In Proceedings of IPC*, 2008.

[28] Justin Carpentier, Rohan Budhiraja, and Nicolas Mansard. Learning feasibility constraints for multi-contact locomotion of legged robots. In *Robotics: Science and Systems*, page 9p, 2017.

[29] 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, 2016.

[30] Rohan Chitnis, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning quickly to plan quickly using modular meta-learning. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7865–7871, 2019.

[31] Rohan Chitnis, Tom Silver, Beomjoon Kim, Leslie Kaelbling, and Tomas Lozano-Perez. CAMPs: Learning context-specific abstractions for efficient planning in factored MDPs. In *Conference on Robot Learning*, pages 64–79. PMLR, 2021.

[32] Rohan Chitnis, Tom Silver, Joshua B Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. GLIB: Efficient exploration for relational model-based reinforcement learning via goal-literal babbling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11782–11791, 2021.

[33] 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.

[34] Jongwook Choi, Yijie Guo, Marcin Moczulski, Junhyuk Oh, Neal Wu, Mohammad Norouzi, and Honglak Lee. Contingency-aware exploration in reinforcement learning. *arXiv preprint arXiv:1811.01483*, 2018.

[35] 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.

[36] Augusto B Corrêa, Florian Pommerening, Malte Helmert, and Guillem Frances. Lifted successor generation using query optimization techniques. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 80–89, 2020.

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

[38] Andrew Cropper and Stephen H Muggleton. Learning higher-order logic programs through abstraction and invention. In *International Joint Conference on Artificial Intelligence*, pages 1418–1424, 2016.

[39] 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.

[40] 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, 2016.

[41] Thomas Dean and Robert Givan. Model minimization in Markov decision processes. In *AAAI/IAAI*, pages 106–111, 1997.

[42] Richard Dearden and Craig Boutilier. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence*, 89(1-2):219–283, 1997.

[43] Thomas Dietterich, George Trimponias, and Zhitang Chen. Discovering and removing exogenous state variables and rewards for reinforcement learning. In *International Conference on Machine Learning*, pages 1262–1270, 2018.

[44] Thomas G Dietterich. State abstraction in MAXQ hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 994–1000, 2000.

[45] Carmel Domshlak and Solomon E Shimony. Efficient probabilistic reasoning in BNs with mutual exclusion and context-specific independence. *International journal of intelligent systems*, 19(8):703–725, 2004.

[46] Danny Driess, Jung-Su Ha, and Marc Toussaint. Deep visual reasoning: Learning to predict action sequences for task and motion planning from an initial scene image. In *Proc. of Robotics: Science and Systems (R:SS)*, 2020.

[47] Danny Driess, Ozgur Oguz, Jung-Su Ha, and Marc Toussaint. Deep visual heuristics: Learning feasibility of mixed-integer programs for manipulation planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

[48] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine learning*, 43(1-2):7–52, 2001.

[49] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

[50] Richard E Fikes, Peter E Hart, and Nils J Nilsson. Learning and executing generalized robot plans. *Artificial intelligence*, 3:251–288, 1972.

[51] 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.

[52] 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.

[53] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 440–448, 2020.

[54] Caelan Reed Garrett, Chris Paxton, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Dieter Fox. Online replanning in belief space for partially observable task and motion problems. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5678–5684, 2020.

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

[56] Daniel Gnad, Alvaro Torralba, Martín Domínguez, Carlos Areces, and Facundo Bustos. Learning how to ground a plan–partial grounding in classical planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7602–7609, 2019.

[57] Paweł Gomoluch, Dalal Alrajeh, and Alessandra Russo. Learning classical planning strategies with policy gradient. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 637–645, 2019.

[58] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[59] Fabien Gravot, Stephane Cambon, and Rachid Alami. aSyMov: a planner that deals with intricate symbolic and geometric problems. In *Robotics Research. The Eleventh International Symposium*, pages 100–110, 2005.

[60] Edward Groshev, Maxwell Goldstein, Aviv Tamar, Siddharth Srivastava, and Pieter Abbeel. Learning generalized reactive policies using deep neural networks. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 2018.

[61] Matthew Grounds and Daniel Kudenko. Combining reinforcement learning with symbolic planning. In *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*, pages 75–86. Springer, 2005.

[62] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468, 2003.

[63] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[64] Dylan Hadfield-Menell, Edward Groshev, Rohan Chitnis, and Pieter Abbeel. Modular task and motion planning in belief space. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4991–4998, 2015.

[65] Patrik Haslum et al. Reducing accidental complexity in planning problems. In *International Joint Conference on Artificial Intelligence*, pages 1898–1903, 2007.

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

[67] 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.

[68] Natalia Hernandez-Gardiol. *Relational envelope-based planning*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2008.

[69] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*, pages 409–426. Springer, 1994.

[70] Jesse Hoey, Robert St-Aubin, Alan J. Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *UAI*, 1999.

[71] Jörg Hoffmann. FF: The fast-forward planning system. *AI magazine*, 22(3):57–57, 2001.

[72] Steven James, Benjamin Rosman, and George Konidaris. Learning portable representations for high-level planning. In *International Conference on Machine Learning*, pages 4682–4691, 2020.

[73] Eric Jang, Alex Irpan, Mohi Khansari, Daniel Kappler, Frederik Ebert, Corey Lynch, Sergey Levine, and Chelsea Finn. BC-Z: Zero-shot task generalization with robotic imitation learning. In *Conference on Robot Learning*, pages 991–1002, 2022.

[74] 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.

[75] Nan Jiang, Alex Kulesza, and Satinder Singh. Abstraction selection in model-based reinforcement learning. In *International Conference on Machine Learning*, pages 179–188, 2015.

[76] Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. A review of generalized planning. *The Knowledge Engineering Review*, 34:e5, 2019.

[77] George H John, Ron Kohavi, and Karl Pfleger. Irrelevant features and the subset selection problem. In *Machine learning proceedings 1994*, pages 121–129. Elsevier, 1994.

[78] Nicholas K Jong and Peter Stone. State abstraction discovery from irrelevant state variables. In *International Joint Conference on Artificial Intelligence*, volume 8, pages 752–757, 2005.

[79] 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.

[80] Michael Kearns, Yishay Mansour, and Andrew Y Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine learning*, 49(2-3):193–208, 2002.

[81] 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.

[82] Beomjoon Kim, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning to guide task and motion planning using score-space representation. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 2810–2817, 2017.

[83] Beomjoon Kim and Luke Shimanuki. Learning value functions with relational state representations for guiding task-and-motion planning. In *Conference on Robot Learning*, pages 955–968, 2020.

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

[85] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[86] Sven Koenig, Maxim Likhachev, Yaxin Liu, and David Furcy. Incremental heuristic search in AI. *AI Magazine*, 25(2):99–99, 2004.

[87] Ron Kohavi and George H John. Wrappers for feature subset selection. *Artificial intelligence*, 97(1-2):273–324, 1997.

[88] Harsha Kokel, Arjun Manoharan, Sriraam Natarajan, Balaraman Ravindran, and Prasad Tadepalli. RePReL: Integrating relational planning and reinforcement learning for effective abstraction. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 533–541, 2021.

[89] Daphne Koller, Nir Friedman, Sašo Džeroski, Charles Sutton, Andrew McCallum, Avi Pfeffer, Pieter Abbeel, Ming-Fai Wong, David Heckerman, Chris Meek, et al. *Introduction to statistical relational learning*. MIT press, 2007.

[90] George Konidaris. Constructing abstraction hierarchies using a skill-symbol loop. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 1648–1654, 2016.

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

[92] 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.

[93] James J Kuffner and Steven M LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 2, pages 995–1001, 2000.

[94] Thanard Kurutach, Aviv Tamar, Ge Yang, Stuart J Russell, and Pieter Abbeel. Learning plannable representations with causal infogan. *Advances in Neural Information Processing Systems*, 31, 2018.

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

[96] Steven M LaValle. *Planning algorithms*. Cambridge University Press, 2006.

[97] Nada Lavrac and Saso Dzeroski. Inductive logic programming. In *Logic Programming Workshop (WLP)*, pages 146–160, 1994.

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

[99] Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *ECAI 2012*, pages 540–545. IOS Press, 2012.

[100] Joao Loula, Kelsey Allen, Tom Silver, and Josh Tenenbaum. Learning constraint-based planning models from demonstrations. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5410–5416, 2020.

[101] 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.

[102] Tengfei Ma, Patrick Ferber, Siyu Huo, Jie Chen, and Michael Katz. Online planner selection with graph neural networks and adaptive scheduling. In *AAAI*, pages 5077–5084, 2020.

[103] 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.

[104] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. *arXiv preprint arXiv:1807.02264*, 2018.

[105] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. *arXiv preprint arXiv:1904.12584*, 2019.

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

[107] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL-the planning domain definition language, 1998.

[108] Neville Mehta, Soumya Ray, Prasad Tadepalli, and Thomas Dietterich. Automatic discovery and transfer of MAXQ hierarchies. In *Proceedings of the 25th international conference on Machine learning*, pages 648–655, 2008.

[109] Alan Miller. *Subset selection in regression*. Chapman and hall/CRC, 2002.

[110] Thomas M Moerland, Joost Broekens, and Catholijn M Jonker. Model-based reinforcement learning: A survey. *arXiv:2006.16712*, 2020.

[111] Thomas M. Moerland, Anna Deichler, Simone Baldi, Joost Broekens, and Catholijn M. Jonker. Think too fast nor too slow: The computational trade-off between planning and reinforcement learning, 2020.

[112] Stephen Muggleton. Inductive logic programming. *New generation computing*, 8(4):295–318, 1991.

[113] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning. In *International Conference on Machine Learning*, volume 1, page 2, 2000.

[114] 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, 2017.

[115] Nils J Nilsson et al. Shakey the robot. *SRI International Technical Report*, 1984.

[116] Michael Noseworthy, Caris Moses, Isaiah Brand, Sebastian Castro, Leslie Kaelbling, Tomás Lozano-Pérez, and Nicholas Roy. Active learning of abstract plan feasibility. *arXiv preprint arXiv:2107.00683*, 2021.

[117] Joaquim Ortiz-Haro, Jung-Su Ha, Danny Driess, and Marc Toussaint. Structured deep generative models for sampling on constraint manifolds in sequential manipulation. In *Conference on Robot Learning*, pages 213–223, 2022.

[118] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[119] 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.

[120] 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, 2016.

[121] Chris Paxton, Vasumathi Raman, Gregory D Hager, and Marin Kobilarov. Combining neural networks and tree search for task and motion planning in challenging environments. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6059–6066, 2017.

[122] Florian Pommerening and Malte Helmert. Incremental LM-cut. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 2013.

[123] David L Poole. Context-specific approximation in probabilistic inference. *arXiv preprint arXiv:1301.7408*, 2013.

[124] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[125] Meng Qu, Yoshua Bengio, and Jian Tang. GMNN: Graph Markov neural networks. *arXiv preprint arXiv:1905.06214*, 2019.

[126] Miguel Ramírez and Hector Geffner. Probabilistic plan recognition using off-the-shelf classical planners. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

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

[128] 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.

[129] Bernardus Ridder. *Lifted heuristics: towards more scalable planning systems*. PhD thesis, King's College London (University of London), 2014.

[130] Pat Riddle, Jordan Douglas, Mike Barley, and Santiago Franco. Improving performance by reformulating PDDL into a bagged representation. In *Proceedings of the 8th Workshop on Heuristic Search for Domain-independent Planning (HSDIP@ ICAPS)*, pages 28–36, 2016.

[131] Or Rivlin, Tamir Hazan, and Erez Karpas. Generalized planning with deep reinforcement learning. *arXiv preprint arXiv:2005.02305*, 2020.

[132] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[133] William Shen, Felipe Trevizan, and Sylvie Thiébaux. Learning domain-independent planning heuristics with hypergraph networks. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 2020.

[134] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[135] Tom Silver and Rohan Chitnis. PDDLGym: Gym environments from PDDL problems. In *International Conference on Automated Planning and Scheduling (ICAPS) PRL Workshop*, 2020.

[136] Tom Silver, Rohan Chitnis, Aidan Curtis, Joshua Tenenbaum, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Planning with learned object importance in large problem instances using graph neural networks. *arXiv preprint arXiv:2009.05613*, 2020.

[137] 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, 2021.

[138] Aravind Srinivas, Ramnandan Krishnamurthy, Peeyush Kumar, and Balaraman Ravindran. Option discovery in hierarchical reinforcement learning using spatio-temporal clustering. *arXiv preprint arXiv:1605.05359*, 2016.

[139] 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, 2014.

[140] Irene Stahl. Predicate invention in ILP—an overview. In *European Conference on Machine Learning*, pages 311–322, 1993.

[141] Kurt Alan Steinkraus. *Solving large stochastic planning problems using multiple dynamic abstractions*. PhD thesis, Massachusetts Institute of Technology, 2005.

[142] Mike Stilman and James J Kuffner. Navigation among movable obstacles: Real-time reasoning in complex environments. *International Journal of Humanoid Robotics*, 2(04):479–503, 2005.

[143] Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[144] Marco Tamassia, Fabio Zambetta, William Raffe, and Xiaodong Li. Learning options for an MDP from demonstrations. In *Australasian Conference on Artificial Life and Computational Intelligence*, pages 226–242, 2015.

[145] Marc Toussaint. Logic-geometric programming: An optimization-based approach to combined task and motion planning. In *International Joint Conference on Artificial Intelligence*, pages 1930–1936, 2015.

[146] 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, 2015.

[147] 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.

[148] Thomas J Walsh. *Efficient learning of relational models for sequential decision making*. PhD thesis, Rutgers, The State University of New Jersey-New Brunswick., 2010.

[149] Zi Wang, Caelan Reed Garrett, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning compositional models of robot skills for task and motion planning. *The International Journal of Robotics Research*, 40(6-7):866–894, 2021.

[150] Andrew M Wells, Neil T Dantam, Anshumali Shrivastava, and Lydia E Kavraki. Learning feasibility for task and motion planning in tabletop environments. *IEEE robotics and automation letters*, 4(2):1255–1262, 2019.

[151] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[152] Fangkai Yang, Daoming Lyu, Bo Liu, and Steven Gustafson. PEORL: Integrating symbolic planning and hierarchical reinforcement learning for robust decision-making. *arXiv preprint arXiv:1804.07779*, 2018.

[153] Jihoon Yang and Vasant Honavar. Feature subset selection using a genetic algorithm. In *Feature extraction, construction and selection*, pages 117–136. Springer, 1998.

[154] Sung Wook Yoon, Alan Fern, and Robert Givan. FF-Replan: A baseline for probabilistic planning. In *ICAPS*, volume 7, pages 352–359, 2007.

[155] Amy Zhang, Rowan McAllister, Roberto Calandra, Yarin Gal, and Sergey Levine. Learning invariant representations for reinforcement learning without reconstruction. *arXiv preprint arXiv:2006.10742*, 2020.

[156] Nevin Lianwen Zhang and David Poole. On the role of context-specific independence in probabilistic inference. In *International Joint Conference on Artificial Intelligence*, volume 1, page 9, 1999.

[157] 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.

[158] Hankz Hankui Zhuo, Derek Hao Hu, Chad Hogg, Qiang Yang, and Hector Munoz-Avila. Learning HTN method preconditions and action models from partial observations. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.