Abstract of “Structured Abstractions for General-Purpose Decision Making” by Cameron Sheehy Allen, Ph.D., Brown University, October 2023.

Generally intelligent agents must learn and plan in complex environments, with sensors and actuators that support various behaviors and tasks. This complexity hinders decision making, necessitating methods that abstract away such intricacies. Such abstractions must be rich enough to be precise, while simple enough to enable efficient learning and planning. Fortunately, many environments contain inherent structure, like Markovian or factored dynamics, that agents can leverage to help manage this tradeoff. By discovering and exploiting the structure in their environment, agents can automatically build decision-making abstractions that are beneficial for learning and planning. This dissertation showcases a collection of methods for abstracting observations, actions, histories, and world models in order to enhance agents’ problem-solving capabilities while striking a useful balance between richness and simplicity.

Structured Abstractions for General-Purpose Decision Making

by
Cameron Sheehy Allen
B. S., Tufts University, 2011
Sc. M., Brown University, 2018

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
October 2023

In memory of Megan Sheehy.

This dissertation by Cameron Sheehy Allen is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

George D. Konidaris, Director

Recommended to the Graduate Council

Date _____ _____

Michael L. Littman, Reader

Date _____ _____

Ronald E. Parr, Reader
Duke University

Approved by the Graduate Council

Date _____ _____

Thomas A. Lewis
Dean of the Graduate School

Vita

Cameron Allen grew up near Boston, Massachusetts. He studied Electrical Engineering and Computer Science at Tufts University and graduated *summa cum laude* with a Bachelor of Science in 2011. After graduating, Cameron worked for four years as an engineer at MITRE, first on communications firmware and later on software, where he received the MITRE Director's Award for engineering excellence. In 2015, Cameron began his Ph.D. in Computer Science at Duke University and completed two semesters there before transferring to continue working with his advisor at Brown University. During his time at Brown, Cameron was a member of the Intelligent Robot Lab and the Brown Integrative General Artificial Intelligence initiative.

Cameron's research was supported by the Air Force Office of Scientific Research Young Investigator Program (grant FA9550-17-1-0124), the National Science Foundation (grants #1717569 and #1955361), the Office of Naval Research (grants N00014-21-1-2200, N00014-22-1-2592, and PERISCOPE Multidisciplinary University Research Initiative N00014-17-1-2699), and by an internship at IBM.

Acknowledgements

I am grateful to so many people for helping to make this dissertation a reality.

First, to my advisor, George Konidaris. Your thoughtfulness and generosity have inspired me since the day we met. I've learned so many things from you, but above all, you taught me to question my deepest assumptions, to always seek out the strongest version of my ideas, and to never lose sight of the bigger picture. Working with you has been a great privilege. Thank you for believing in me.

To my committee members, Michael Littman and Ron Parr, for reminding me to approach the world with curiosity and humor. I aspire to match your seemingly inexhaustible supply of creative ideas and insightful questions.

To my earlier research mentors: Gerry Tesauro, for sharing the wisdom that impactful research requires uncertainty about the outcome; Ron Lasser, for teaching me to keep asking questions until the true problem finally reveals itself; and Eric Miller, for introducing me to research in the first place and providing crucial words of initial encouragement.

To my many collaborators, for your brilliance, drive, and collective role in shaping the work presented in this dissertation. You made the moments of understanding much richer and the moments of frustration much more tolerable.

To my colleagues at Brown, Duke, IBM, and around the world, for your interest in my ideas and for graciously sharing yours. Thank you for making me feel welcome.

To my friends and family, for your kindness and support during my moments of self-doubt and your insistence on celebrating even the smallest victories. And in particular, to my parents, Lisa and Bill, and my brother, Graham, for challenging me to dive deeper and climb higher.

And finally, to everyone who I did not mention by name, but who nevertheless supported me on this epic journey of scientific and self-discovery. Thank you all so much.

Contents

1 Introduction **1**

2 Background **3**

2.1 Reinforcement Learning 4

2.1.1 Markov Decision Processes 4

2.1.2 General Decision Processes and Partial Observability 8

2.1.3 Block MDPs 10

2.1.4 Factored MDPs 11

2.1.5 Model-Based RL 12

2.2 Planning 13

2.2.1 Classical Planning 14

2.2.2 Numeric Planning 17

2.2.3 Black-Box Planning 18

2.2.4 Open-Scope Planning 19

2.3 Abstraction 19

2.3.1 Observation Abstractions 20

2.3.2 History Abstractions 22

2.3.3 Action Abstractions 23

2.3.4 World-Model Abstractions 24

3 Observation Abstractions **26**

3.1 Related Work 27

3.1.1 Bisimulation 27

3.1.2 Ground-State Prediction and Reconstruction 29

3.1.3 Inverse Dynamics Models 29

3.1.4 Contrastive Learning 29

3.1.5 Kinematic Inseparability 30

3.1.6 Other Approaches 30

3.2 Markov State Abstractions 31

3.2.1 Sufficient Conditions for a Markov Abstraction 32

3.2.2 An Inverse Model Counterexample . . . 33
3.3 Training a Markov State Abstraction . . . 33
3.3.1 Inverse Models . . . 33
3.3.2 Density Ratios . . . 34
3.3.3 Smoothness . . . 34
3.3.4 Markov Abstraction Objective . . . 35
3.4 Offline Abstraction Learning for Visual Gridworlds . . . 36
3.5 Online Abstraction Learning for Continuous Control . . . 36
3.6 Conclusion . . . 39

4 History Abstractions **40**
4.1 Learning Memory via the λ-Discrepancy . . . 41
4.1.1 The λ-Discrepancy . . . 43
4.1.2 When is the λ-discrepancy zero? . . . 44
4.2 Memory Optimization . . . 45
4.2.1 Analytical Memory Optimization . . . 45
4.2.1.1 Learning Memory Functions . . . 46
4.2.1.2 Analytical Memory Optimization Planning Experiments . . . 47
4.2.2 Sample-Based Memory Optimization Learning Experiments . . . 48
4.3 Conclusion . . . 49

5 Action Abstractions **50**
5.1 Goal-Count Accuracy and Effect Size . . . 51
5.1.1 The Suitcase Lock Domain . . . 52
5.1.1.1 Focused Actions Improve the Goal-Count Heuristic . . . 52
5.1.1.2 Focused Actions Improve Planning Efficiency . . . 53
5.2 Learning Macros with Focused Effects . . . 53
5.3 Experiments . . . 55
5.3.1 Methodology . . . 55
5.3.2 Comparisons with Other Planners . . . 56
5.3.3 Comparison with Random Macros . . . 57
5.3.4 Examining Expert Macros in Rubik's Cube . . . 59
5.3.5 Interpretability of Focused Macros . . . 59
5.3.6 Generalizing to Novel Goal States . . . 60
5.4 Related Work . . . 60
5.5 Conclusion . . . 61

6 World-Model Abstractions **62**
6.1 Background . . . 63
6.1.1 Defining Task-Relevance . . . 64

6.2 Task Scoping . . . 66
6.2.1 Backwards Reachability of Variables . . . 66
6.2.2 Merging Same-Effect Operators . . . 67
6.2.3 Causally Linked Irrelevance . . . 68
6.2.4 Main Theorem . . . 68
6.2.5 Discussion . . . 70
6.3 Experimental Evaluation . . . 70
6.3.1 Numeric Domains with ENHSP . . . 71
6.3.2 Classical Planning with Fast Downward . . . 73
6.4 Related Work . . . 74
6.5 Conclusion . . . 75

7 Closing Remarks **76**

Appendix **78**

A Markov State Abstractions **79**
A.1 Limitations . . . 79
A.2 Glossary of Symbols . . . 80
A.3 General-Policy Definitions . . . 81
A.3.1 General-Policy Definitions . . . 81
A.4 Proofs . . . 82
A.4.1 Main Theorem . . . 82
A.4.2 Inverse Model Implies Density Ratio . . . 85
A.5 Derivation of Density Ratio Objective . . . 86
A.6 Markov State Abstractions and Kinematic Inseparability . . . 87
A.6.1 Example MDP . . . 87
A.6.2 "Strongly Markov" Implies KI . . . 88
A.7 Implementation Details for Visual Gridworld . . . 90
A.7.1 Computing Resources . . . 90
A.7.2 Network Architectures . . . 90
A.7.3 Hyperparameters . . . 91
A.8 Implementation Details for DeepMind Control . . . 92
A.8.1 Computing Resources . . . 92
A.8.2 Markov Network Architecture . . . 92
A.8.3 Hyperparameters . . . 93
A.9 Additional Representation Visualizations . . . 94
A.10 Gridworld Results for Increased Pretraining Time . . . 95
A.11 DeepMind Control Experiment with RBF-DQN . . . 96
A.12 DeepMind Control Ablation Study . . . 97

B Lambda Discrepancy **98**
B.1 TD(λ) Fixed Point . . . 98
B.2 Proof of Theorem 2 (Almost All) . . . 101
B.3 Proof of Lemma 4.1.1 (Block MDP) . . . 102
B.4 Proof of Lemma 4.1.2 (Zero λ-Discrepancy) . . . 103
B.5 Lambda Discrepancy Norm Weighting . . . 104
B.6 Analytical Memory Optimization . . . 104
B.6.1 Memory-Augmented POMDP . . . 104
B.6.2 Environments and Analytical Algorithms . . . 105
B.6.2.1 T-Maze Details . . . 105
B.6.2.2 Other POMDP Details . . . 106
B.6.2.3 Memory Improvement Algorithm . . . 106
B.6.3 Analytical Memory Cartesian Product . . . 107
B.6.4 Memory Optimization for Value Improvement . . . 108
B.6.5 Experiment Details . . . 108
B.6.5.1 Analytical Memory Optimization Experiment Details . . . 108
B.6.5.2 Discussion of Tiger Results . . . 109
B.7 Sample-Based Experiments . . . 109
B.7.1 Sample-Based Results on Tabular POMDPs . . . 109
B.7.2 Sample-Based Results on RockSample . . . 110
B.7.3 Experimental details for RockSample . . . 110

C Focused Macro-Actions **114**
C.1 Suitcase Lock Implementation . . . 114
C.2 Simulator Details . . . 115
C.2.1 PDDLGym . . . 115
C.2.2 15-Puzzle . . . 115
C.2.3 Rubik's Cube . . . 116
C.3 Updating the Simulator with Macros . . . 117
C.4 Expert Rubik's Cube Macros . . . 119
C.5 Reproducibility . . . 119
C.5.1 Hyperparameter Selection . . . 119
C.5.1.1 PDDLGym Domains . . . 119
C.5.1.2 15-Puzzle . . . 120
C.5.1.3 Rubik's Cube . . . 120
C.5.2 Computational Resources . . . 120
C.5.3 Random Seeds . . . 120

D Task Scoping **121**
D.1 Results using Fast Downward's Merge and Shrink Heuristic . . . 121

D.2 Detailed Experimental Domain Descriptions . . . 121
D.2.1 Numeric Planning Domains . . . 122
D.2.1.1 Multi-Switch Continuous Playroom . . . 122
D.2.1.2 Composite IPC Domain . . . 122
D.2.1.2.1 Satellites . . . 123
D.2.1.2.2 Driverlog . . . 123
D.2.1.2.3 Depots . . . 123
D.2.1.2.4 Zenotravel . . . 123
D.2.1.3 Minecraft . . . 123
D.2.2 IPC Domains . . . 124
D.2.2.1 Logistics . . . 124
D.2.2.2 DriverLog . . . 125
D.2.2.3 Satellite . . . 125
D.2.2.4 ZenoTravel . . . 125
D.3 Full PDDL for example domain . . . 126
D.3.1 Domain File . . . 126
D.3.2 Problem File . . . 127

⋆ Parts of this thesis were conducted in collaboration with others and have been previously published or presented elsewhere.

- Chapter 3 is based on work led by myself, conducted with Neev Parikh, Omer Gottesman, and George Konidaris, and published in *Advances in Neural Information Processing Systems*, volume 34, December 2021. Copyright the authors.
- Chapter 4 is based on work co-led by myself, Aaron Kirtland and Ruo Yu Tao, conducted with Daniel Scott, Sam Lobel, Nicholas Petrocelli, Omer Gottesman, Michael L. Littman, and George Konidaris, which has been submitted for publication and is currently under review. Copyright the authors.
- Chapter 5 is based on work led by myself, conducted with Michael Katz, Tim Klinger, George Konidaris, Matthew Riemer, and Gerald Tesauro, and published in *Proceedings of the 30th International Joint Conference on Artificial Intelligence*, August 2021. Copyright International Joint Conferences on Artificial Intelligence, Inc. (IJCAI).
- Chapter 6 is based on work led by Michael Fishman and Nishanth Kumar, conducted with myself, Natasha Danas, Michael L. Littman, Stefanie Tellex, and George Konidaris, and presented at the IJCAI Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning, August 2023. Copyright the authors.

Structured Abstractions for General-Purpose Decision Making

Cameron Sheehy Allen

October 2023

Chapter 1

Introduction

> A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects.
>
> —Robert A. Heinlein

Any intelligent software agent requires sufficient sensors for making decisions. The sensors supply the agent with information on which to base its current and future actions. For example, a cleaning robot must know where it should clean and be able to verify that it has successfully done the job. A self-driving car must be aware of the road ahead, the immediate surroundings, the directions to the next destination, and the physical state of the car itself. A chess-playing robot must know the current board position, whose turn it is, and the size and positioning of its own arms for moving pieces around. Without the appropriate sensors, an agent cannot make decisions effectively.

Moreover, carrying out decisions requires interacting with the environment in the appropriate way. The set of actions worth considering depends on the problem being solved. For instance, the cleaning robot should be capable of moving itself around, picking up objects, and manipulating vacuums and mops. The self-driving car should be able to control the car's steering, acceleration, and braking. The chess-playing robot should know how and when the different pieces are allowed to move. Just as each task has a relevant set of sensor data, there is also a corresponding set of actions that best characterize the decision-making problem.

General-purpose agents therefore require general-purpose sensors and diverse action capabilities. Not every problem depends on the same sensor information: cleaning status is irrelevant for chess; board positions

are irrelevant for driving a car; and steering wheel angle is irrelevant for cleaning. And the set of relevant behaviors changes as well: from selecting a vacuum speed, to changing highway lanes, to castling queenside. Nevertheless, a single robot may one day be asked to clean a home, drive a car, *and* play a game of chess. To support general-purpose decision making, the sensor observations must be sufficiently rich, and the action set sufficiently flexible, for any task the agent may encounter.

Unfortunately, generality leads to complexity. For an agent to be capable of tackling a wide range of problems, it must necessarily be over-engineered for any one of them. The rich observations and low-level actions of general-purpose agents are not ideal for most problems. Imagine trying to play a game of chess by reasoning about the pixels in your video feed, rather than the locations of the pieces—or plan a road trip as a sequence of steering wheel and motor torques, rather than a list of roads and cities. Without some way to mitigate the inherent complexity that comes with generality, decision making quickly becomes intractable.

Effective general-purpose decision making therefore requires problem-specific abstractions. Abstractions encode and reveal problem structure to simplify the decision-making process. They can transform collections of pixels into abstract concepts—items of clothing, lanes and roads, chessboard squares—and in so doing, restrict the available sensor information to only what is important. They ease action selection by introducing task-relevant skills such as shirt folding, lane following, and knight movement. Appropriate abstractions enable high-level decision making on top of low-level sensors and actions.

Since what is relevant changes with each problem, the process of finding a suitable abstraction must be automatic. While, in principle, a system designer could specify a suitable problem-specific abstraction by hand, this is unsatisfying for several reasons. First, specifying an abstraction manually can be time-consuming, error-prone, and highly technical. Second, the designer may not actually know what a suitable abstraction would be for a given problem. Third, the set of tasks the agent may face is infinite, so clearly it is infeasible to manually specify an abstraction for every possible task. Finally, excessive reliance on external guidance reduces the agent's autonomy and severely limits its usefulness as an independent decision-making system.

The goal of this dissertation is to develop methods by which general-purpose agents can leverage structure in their environments to autonomously construct suitable abstractions for high-level decision-making. The content will be divided into four main parts. First, we consider the problem of abstracting the agent's rich sensor *observations* while preserving sufficient information to fully characterize the environment's dynamics at every decision point. Then, we discuss abstractions over *histories* that exploit structure in the agent's evaluations of its own behavior in order to determine what it should remember. Next, we turn to *action* abstraction and describe how to build a collection of high-level skills that tease apart the different environmental aspects in the agent's internal world model. Finally, we look at abstractions over *world models* themselves, which focus the agent's attention on only the skills relevant for solving its current task. Collectively, the dissertation will provide evidence for the thesis that by discovering and exploiting the structure in their environment, agents can autonomously build decision-making abstractions that are beneficial for learning and planning.

Chapter 2

Background

> All models are wrong, but some are useful.
>
> In applying mathematics to subjects such as physics or statistics we make tentative assumptions about the real world which we know are false but which we believe may be useful nonetheless.
>
> —George E. P. Box

Our aim is for agents to automatically construct their own decision-making abstractions across a range of problems and contexts. Achieving this level of generality requires distilling the essential aspects of decision making into a common conceptual framework, and then designing strategies that are broadly effective over the set of problems that framework can express. This, in itself, is a form of abstraction.

There are many ways to formalize the problem of sequential decision making. Here, we consider two of the most common: reinforcement learning and planning. These two approaches differ in the assumptions that they make, but they are—broadly speaking—mutually compatible. In reinforcement learning, tasks are typically specified using a reward signal, and the agent's job is to change its behavior so as to generate more reward. Planning tasks are instead specified as goal conditions, and the agent's job is to find a viable strategy for achieving a particular goal. Additionally, the planning setting typically assumes the agent has access to an accurate internal world model that it can leverage when considering the effects of hypothetical actions, whereas in the learning setting, such a model must be constructed through experience (if at all).

With these decision-making formalisms in place, we then turn our attention to formalizing the process of abstraction. We consider the two most essential aspects of learning and planning—observations and actions—and define abstraction in each context. We use abstraction to reduce complexity and transform large observation and action spaces into smaller ones. We also consider the reverse—*augmenting* observations and actions—and

show how these transformations can *also* be viewed as abstractions, only now over sequences of past and future experiences. Together, these complementary forms of abstraction provide us with a flexible set of tools for building agents that adapt to different problem-solving scenarios.

2.1 Reinforcement Learning

Reinforcement learning (RL) makes the relationship between an agent and its environment explicit. The agent's sensors define its observations, and its actions define how it can affect the environment. The decision process is represented as a loop (see Figure 2.1): the agent takes an action, thereby affecting the environment, and subsequently observes the result through its sensors (along with any reward information), at which point the agent selects another action and the process repeats.

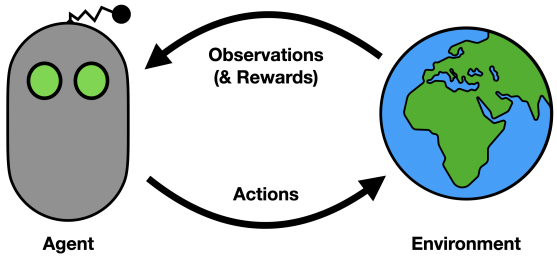


Figure 2.1: The reinforcement learning framework.

Reinforcement learning is an incredibly flexible framework and can easily be customized for specific applications or to encode various types of assumptions or domain knowledge. We begin with the most basic model, the Markov decision process (MDP), in which observations are assumed to contain complete state information about every aspect of the agent-environment system. Next, we discuss more general decision processes and the challenges that arise when the agent's observations are incomplete. Then we briefly mention two useful sub-classes of MDPs, block MDPs and factored MDPs, which make additional structural assumptions. Finally, we will ease the transition from reinforcement learning to planning by touching on hybrid approaches where the agent learns an internal model of the world to aid in its decision making.

2.1.1 Markov Decision Processes

The Markov decision process[1] is one of the simplest formal models of sequential decision making. Its simplicity stems from an assumption, the *Markov assumption*, that each of the agent's observations contains complete state information about the agent-environment system. This assumption makes it rather easy to show that the agent can make optimal decisions simply by conditioning its behavior on its current observation.

A Markov decision process consists of the following quantities:

[1] Named after Russian mathematician Andrey Andreyevich Markov (Markov, 1906; Seneta, 1996).

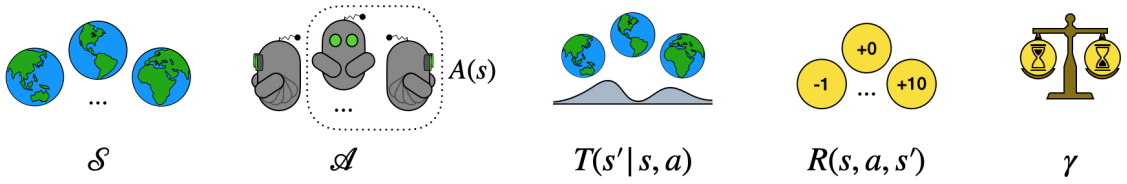


Figure 2.2: Components of a Markov decision process.

- A set of states, $\mathcal{S}$, where each state s corresponds to one possible configuration of the agent-environment system;
- A set of actions, $\mathcal{A}$, and an action applicability function, $A(s) \subseteq \mathcal{A}$, that outputs the set of valid actions that the agent can select in a given state s;[2]
- A transition function, $T(s'|a, s)$, measuring the probability of transitioning to state s' after taking action a from state s;
- A reward function, $R(s, a, s')$ that assigns a reward r to each combination of state s, action a, and next-state s';
- A discount factor, $\gamma \in [0, 1]$ quantifying how much less the agent values rewards when they are delayed by one additional time step.

At each timestep, the agent observes state s_t, takes action a_t from among $A(s_t) \subseteq \mathcal{A}$, then the system transitions to some state s_{t+1} sampled from $T(s_{t+1}|a_t, s_t)$, at which point the agent receives reward $r_t = R(s_t, a_t, s_{t+1})$ and observes state s_{t+1}. By convention, the system is initialized at timestep $t = 0$.

To formalize the Markov assumption, we simply mean that the state representation exhibits the *Markov property*, namely that transitions $T(s'|a, s)$, rewards $R(s, a, s')$, and valid actions $A(s)$ are all independent of any state history before the most recent state s.

Definition 1 (Markov State Representation). *A decision process* $M = (\mathcal{S}, \mathcal{A}, A, R, T, \gamma)$ *and its state representation* $\mathcal{S}$ *are* Markov *if and only if:*

$$A^{(k)}\left(s_t, a_{t-1}, s_{t-1}, \ldots, a_{t-k}, s_{t-k}\right) = A(s_t),$$

$$R^{(k)}\left(s_{t+1}, a_t, s_t, a_{t-1}, s_{t-1}, \ldots, a_{t-k}, s_{t-k}\right) = R(s_{t+1}, a_t, s_t),$$

and

$$T^{(k)}\left(s_{t+1}|a_t, s_t, s_t, a_{t-1}, s_{t-1}, \ldots, a_{t-k}, s_{t-k}\right) = T(s_{t+1}|a_t, s_t),$$

for all $a \in \mathcal{A}$, $s \in \mathcal{S}$, $k \geq 1$.

The superscript (k) denotes that the function is being conditioned on k additional steps of history.

[2] It's common to assume that $A(s) = \mathcal{A}$ for all states and that the actions that would otherwise be inapplicable simply have no effect.

Example 1 (Pendulum). Suppose we want to construct a Markov decision process to represent a decision problem for controlling a pendulum (see Figure 2.3).

The pendulum contains a motor inside the hinge, which can supply three possible torques: $\{-\tau, 0, +\tau\}$. The agent's task is to apply torques in order to keep the pendulum balanced within some small deviation from fully upright.

Perhaps we decide to use the counterclockwise pendulum angle θ as the state space: $\mathcal{S} := \theta \in [-\pi, \pi)$, with reference angle $\theta = 0$ corresponding to fully vertical.

We choose the three distinct torque values as the natural action space, and set: $\mathcal{A} := \{-\tau, 0, +\tau\}$. All actions are applicable for every state, so $A(s) := \mathcal{A}$.

We then define rewards R as follows:

$$R(s,a) := \begin{cases} +1, \text{ if } |\theta| < \epsilon \\ 0, \text{ otherwise.} \end{cases}$$

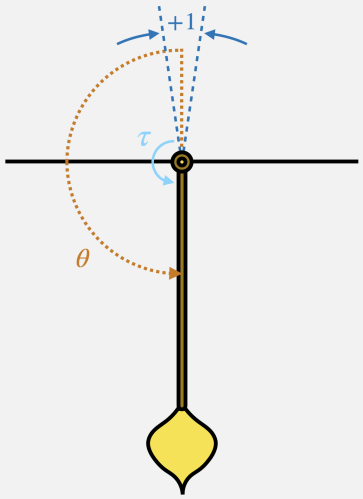


Figure 2.3: A controlled pendulum.

Now we must define the transition function T, and here we encounter a problem. How can we describe the resulting pendulum angle in terms of the current angle and the applied torque? Our prediction seems to depend on whether the pendulum is stationary or swinging in from either side with some angular velocity. In other words, the system isn't Markov. We can't precisely define the transition function in terms of the current state without additional history information.

But suppose we had instead chosen a different state space, consisting not only of pendulum angle θ, but also $\dot{\theta}$, its angular velocity: $\mathcal{S} := \theta \in [-\pi, \pi); \dot{\theta} \in \mathbb{R}$.

Keeping $\mathcal{A}$ and R as defined above, we can now define the transition function T as follows:

$$T(s'|a,s) := \begin{cases} \theta \mathrel{+}= \dot{\theta} \cdot dt \\ \dot{\theta} \mathrel{+}= \left(\frac{g}{2L}\sin(\theta) + \frac{a}{mL^2}\right) \cdot dt, \end{cases}$$

where g is the acceleration due to gravity, L and m are the length and mass of the pendulum, and dt is the amount of time between timesteps.

The resulting decision process can be specified precisely in terms of the state $s = (\theta, \dot{\theta})$, without any additional history information, and is therefore a Markov decision process.

The behavior of a reinforcement learning agent is typically determined by a policy $\pi : \mathcal{S} \to \Pr(\mathcal{A})$, mapping from states to distributions over actions. Each policy induces a value function, $V_\pi : \mathcal{S} \to \mathbb{R}$, and an action-value function, $Q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, which are defined as the expected discounted sum of future rewards (also known as the *return*, denoted G_t) starting from a given state (and action) and following the policy π thereafter:

$$V_\pi(s) := \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \middle| s_t = s;\ T,\ R\right] = \mathbb{E}_\pi\left[G_t | s_t = s;\ T,\ R\right],$$
$$Q_\pi(s,a) := \mathbb{E}_\pi\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \middle| s_t = s, a_t = a;\ T,\ R\right] = \mathbb{E}_\pi\left[G_t | s_t = s, a_t = a;\ T,\ R\right],$$

where γ is typically chosen to be less than 1 so that these sums remain finite. An alternative, but equivalent, interpretation is that $(1-\gamma)$ represents the *termination probability*, i.e. the probability in each timestep that the agent becomes distracted (or breaks, dies, etc.), resulting in what's called a *terminal state*. Under that interpretation, the value function is simply the expected total reward after accounting for these terminations (Puterman, 1994).

The agent's objective is to learn an optimal policy π^* that maximizes value at every state. Such a policy always exists for an MDP precisely because of the Markov property (Puterman, 1994). In other words, the Markov property is desirable because it means the state perfectly characterizes the system, and thus the agent can behave optimally without considering history information. Common strategies for learning an optimal policy include: estimating an action-value function and selecting actions that appear to have higher value; and/or maintaining an explicit policy and improving it over time based on reward feedback. A full discussion of learning algorithms is outside the scope of this dissertation, but see Sutton and Barto (2018) for a detailed treatment.

The Markov property also has a desirable implication for value functions: a recursive relationship known as the *Bellman equation* (Bellman, 1954). The Bellman equation relates the value of the agent's policy at one timestep to the value at the following timestep:

$$\begin{aligned} V_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} T(s'|a,s)\Big(R(s,a,s') + \gamma V_\pi(s')\Big); \\ Q_\pi(s,a) &= \sum_{s' \in \mathcal{S}} T(s'|a,s)\Big(R(s,a,s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') Q_\pi(s',a')\Big). \end{aligned} \tag{2.1}$$

The Bellman equation allows agents to estimate a value function through an iterative update process called temporal difference (or *TD*) learning (Sutton, 1988). Given an experience $(s \xrightarrow{a} (r, s'))$ in the environment, the agent can update its estimated value function to minimize the *temporal difference error* between its current value estimate $\hat{V}_\pi(s)$ and the one-timestep "unrolled" version $(r + \gamma \hat{V}_\pi(s'))$:

$$\hat{V}_\pi^{i+1}(s) := \hat{V}_\pi^i(s) + \alpha\Big(r + \gamma \hat{V}_\pi^i(s') - \hat{V}_\pi^i(s)\Big),$$

where $\alpha \in (0, 1]$ is a step-size parameter that controls how quickly to update $\hat{V}_\pi$, and $\hat{V}_\pi^0$ is initialized arbitrarily. We often refer to this process as *bootstrapping*, since we use our current estimate $\hat{V}_\pi^i$ to improve our next estimate $\hat{V}_\pi^{i+1}$. After a sufficient number of iterations, $\hat{V}_\pi$ will eventually converge[3] to the true value function V_π. The same procedure is compatible with learning Q-functions as well.

[3] Under certain modest assumptions that ensure the agent sees enough data; see Puterman (1994) for more detail.

Temporal difference learning has a large number of variations. We can replace the 1-step unrolling $(r+\gamma\hat{V}_\pi(s'))$ with any valid estimate of the expected return. For example, an n-step unrolling $(r_t+\gamma r_{t+1}+\cdots+\gamma^n\hat{V}_\pi(s_{t+n}))$ leads to n-step TD. As n approaches infinity (or, practically speaking, if we simply let episodes terminate naturally and collect all the discounted rewards without any bootstrapping), we obtain what's called a *Monte Carlo* value estimate. And any weighted sum of these estimates is a valid estimate as well. If we downweight successive unrollings geometrically, we obtain a class of methods called TD(λ):

$$\hat{V}_\pi^{i+1}(s_t) := \hat{V}_\pi^i(s_t) + \alpha(1-\lambda)\Big(\sum_{n=1}^{\infty}\lambda^{n-1}\big(r_t+\cdots+\gamma^n\hat{V}_\pi^i(s_{t+n})\big) - \hat{V}_\pi^i(s_t)\Big),$$

where $\lambda \in [0,1]$ controls how quickly we decay the weights. For the special cases of $\lambda = 0$ and $\lambda = 1$, we recover the basic 1-step TD and Monte Carlo value estimation, respectively.

2.1.2 General Decision Processes and Partial Observability

The Markov assumption is, of course, totally unrealistic for general-purpose intelligent agents. A cleaning robot may need to clean multiple rooms, with sensors that can't see past the nearest walls. Similarly, a self-driving car must account for the possibility of cars beyond its field of view. Even a chess-playing robot cannot always determine whose turn it is simply by looking at the board. While the Markov decision process is a useful formalism for some applications, it falls short of being truly general.

We can build a more general formalism by introducing so-called *partial observability*. A *partially observable Markov decision process* (or POMDP) builds in uncertainty about which state the agent is in by separating the agent's observations from the agent-environment state (Kaelbling et al., 1995). In the POMDP model, we add the following quantities to the basic MDP outlined in the previous section:

- A set of possible observations, Ω, where each observation ω corresponds to a particular sensor measurement the agent might see;
- An observation function, $O(\omega|s)$, measuring the probability of seeing a particular observation ω, conditioned on being in state s.

By distinguishing between the complete state of the system, s, and the agent's possibly-incomplete observation, ω, we are now able to model the effects of the agent's sensors. When $\omega = s$ for all s, this formalism reduces to a fully observable MDP.

POMDPs offer an appealing and general model of decision making, but they also present a number of challenges. Since the agent no longer knows the precise state, it cannot rely on the Markov property to simplify decision making. The agent can of course make the Markov assumption anyway, simply pretending its observations are complete, but it will frequently be wrong, and this will lead to making worse decisions. Alternatively, the agent can condition its behavior on its entire interaction history of observations, actions, and rewards. By definition, this would incorporate all information the agent could possibly use for decision making, and hence would constitute a Markov state representation. The problem with this approach is that

the length of the agent's history grows with each successive decision, and the number of possible histories is exponential in that history length. So even *writing down* the agent's policy for all of those histories quickly becomes intractable.

Feature Augmentation. One of the simplest strategies for decision making in a POMDP is to have a human expert procedurally augment the natural observation space to render it Markov. For example, in Atari video games (Bellemare et al., 2013), the current observation is usually *frame-stacked*—concatenated with three previous frames—to include information about the velocity and acceleration of moving objects (Mnih et al., 2015). Applications ranging from automated HVAC control (Galataud, 2021) to stratospheric balloon navigation (Bellemare et al., 2020) use task-specific state features hand-engineered to approximate the Markov assumption. While these applications are impressive, these hand-designed features lack generality across environments.

Belief State Methods. Another common strategy for solving POMDPs is to estimate a *belief distribution*, $b(s)$, which measures the probability that the agent is in a particular state s, given its entire history. The agent can then select actions to maximize its expected value over all states, weighted by the belief distribution. Since belief distributions condition on the full history, this approach might seem hardly any better than the naïve solution. But it turns out that the belief distribution itself constitutes a proper Markov state (Kaelbling et al., 1995). If we know the belief distribution $b(s)$ at some timestep, we can write the next belief distribution $b(s')$, after taking action a and observing ω', as follows:

$$b(s') = \frac{1}{\eta} O(\omega'|s') \sum_{s \in \mathcal{S}} T(s'|a, s) b(s), \tag{2.2}$$

where $\eta = \sum_{s' \in \mathcal{S}} O(\omega'|s') \sum_{s \in \mathcal{S}} T(s'|a, s) b(s)$ is a normalizing constant so that the probabilities sum to 1. Equation (2.2), known as the *belief update*, looks at each state s in our current belief distribution and considers how likely it is for action a to result in state s', along with how likely it is to then observe ω' in that state, and uses this information to construct an updated belief for each possible s'. The distribution $b(s)$ is often called a *belief state*, since it is sufficient for predicting $b(s')$.

Unfortunately, belief state methods rely on some rather strong assumptions. In particular, the belief update requires knowing the dynamics of the environment and sensor, as well as the set of possible states, despite all of these being inherently unobservable. It's more appropriate to categorize such methods as *planning* methods, which we'll discuss in Section 2.2, since they assume an accurate internal world model. In order to qualify as a *learning* method, an agent would not only need to build a model of the environment and sensor dynamics from experience, but also somehow infer the set of hidden states without ever observing them. In the learning setting, POMDP difficulty scales with the number of possible histories, and belief state methods are intractable for all but the smallest problems (Zhang et al., 2012). The POMDP model has mainly led to success in domains like robotics (Thrun et al., 2005), where it is often reasonable for the robot to know the latent state space $\mathcal{S}$ and thus employ belief-state approaches.

Predictive State Representations. An alternative approach to solving POMDPs is to construct a *predictive state representation* (PSR) that can still be updated recursively while requiring only observable quantities

(Littman et al., 2001). Rather than looking at histories, PSRs are collections of tests about *future* action-observation sequences. Each test (for example, "If I keep driving forward, will the traffic light be red?") captures the probability that the observation (red light) occurs after selecting the action (drive forward), and is associated with some likelihood (say 40%). Tests that span multiple time steps (like forward–yellow–forward–red) correspond to the probability of seeing the entire observation sequence (yellow light, followed by red) while following the entire action sequence (driving forward for two timesteps). Given a sufficiently diverse and descriptive set of tests, predictive state representations constitute a Markov state and are expressive enough to represent any POMDP (Singh et al., 2004b).

But while PSRs are appealing in theory, there are three main challenges to applying them in practice. First, finding a general method for constructing the set of tests remains an open problem. Most approaches still require looking at the hidden state information to do so, with the tests designed (often by hand) to distinguish between those hidden states (Singh et al., 2003). Second, even once a viable set of tests has been established, the problem of estimating their probabilities remains, which is especially challenging because tests can involve mutually exclusive sequences of actions. If test A requires the "turn left" action and test B requires the "turn right" action, only one of these can actually be executed in any particular circumstance. Third, estimation of the test probabilities may require many evaluations before the estimates are accurate, but returning to the "same state" (in order to take multiple measurements) is fraught, given that the agent's only concept of state relies on those very same estimates. As a result, predictive state representations have seen limited success outside the planning setting, apart from a few specific examples (Boots et al., 2011; Boots et al., 2013).

Other Methods

There are several other approaches to handling partial observability. Hutter's (2016) extreme state aggregation aims to form a small, stationary MDP from a potentially non-Markov system. Recurrent neural networks (RNNs) trained via Backpropagation Through Time (BPTT) can work well on some environments (Lin and Mitchell, 1993; Bakker, 2001), but their success is greatly affected by the truncation window, which limits how long they can recall information. Additionally, some work has tried to combine RNNs with PSRs (Downey et al., 2017). General value functions (GVFs), implemented such as through the Horde algorithm (Sutton et al., 2011), aim to make predictions about the environment that can encompass partially observable information. A further implementation of GVFs with RNNs, called GVF Networks (Schlegel et al., 2021), can, in fact, represent PSRs. Though GVF-based methods can be made to work in particular instances, they tend to fail in general.

2.1.3 Block MDPs

To make some progress towards modeling rich, low-level sensor observations while avoiding the complexity of partial observability, Du et al. (2019) introduced a hybrid model called the *block MDP*. Block MDPs allow hidden states to produce many different observations, and these observations may be much more detailed than necessary to characterize the environment state; however, observations that come from the same "block"

always correspond to the same underlying state.

Formally, block MDPs make the following assumption to remove partial observability:

Block MDP Assumption: For any two distinct states $s_1 \neq s_2$, the corresponding observation distributions $O(\omega|s_1)$ and $O(\omega|s_2)$ have non-overlapping support.

This assumption encodes a strong structural constraint on the observation function, namely that two different states cannot produce the same observation. Consequently, there exists an inverse observation function $O^{-1} : \omega \mapsto s$ (possibly unknown to the agent) that uniquely identifies the complete state s for any observation ω. In other words, the agent's observations are themselves complete, and ω can be used in place of s without requiring additional history. As with POMDPs, the learning agent doesn't observe the hidden state s, observation function O (nor its inverse O^{-1}), reward function R, nor environment dynamics T.

Block MDPs provide a useful starting point for modeling general-purpose sensors, and we will use them in Chapter 3 to explore observation abstractions, before returning to the full generality of POMDPs in Chapter 4 with history abstractions.

2.1.4 Factored MDPs

Whereas block MDPs impose structural constraints on the observation function, *factored MDPs* (Boutilier et al., 1999; Koller and Parr, 1999) use structure to constrain the transition function. The factored MDP model introduces a formal notion of state variables, providing a way to characterize the causal relationships between the different aspects of the decision process. Rather than a single, monolithic transition function governing the entire state of the system, these state variables allow for a combination of multiple, weakly-interacting transition functions that each affect a smaller portion of that state, and whose individual dynamics are (hopefully) easier to learn.

The factored MDP framework reflects two complementary intuitive principles. First, actions tend to have focused effects: general-purpose agents can affect many aspects of their environments, but cannot affect the majority of those aspects at the same time. Second, effects tend to have focused causes: most aspects of the system behavior depend on only a small subset of the available state information. These principles apply equally well to both partially- and fully observable decision making, but we will focus on the fully observable version, which requires the following modifications to the basic MDP model:

- Each state s is expressed as a (finite) vector of K state variables or *factors*, where each variable s_i is assigned a value from some domain set D_i, such that $S = D_1 \times D_2 \times \cdots \times D_K$.
- Each state variable s_i is governed by a separate transition function $T_i(s'_i|a, s)$. Moreover, this transition function only depends on a subset of the state variables, denoted by a *parents* function $\rho(s_i)$, and conditioning the next-state variable s'_i on these parents makes it independent of all other state variables: $T_i(s'_i|a, s) = T_i(s'_i|a, \rho(s'_i))$.

- The full transition function $T(s'|a, s)$ factorizes as the product of these K separate transition functions: $T(s'|a, s) = \Pi_{i=1}^{K} T_i(s'_i|a, \rho(s'_i))$.

These modifications alone merely allow for notational convenience, because it is always possible to factor the transition function, provided that the state is composed of multiple distinct variables. Therefore, we typically make one more assumption with factored MDPs, namely that the maximum number of parents $\mathcal{P} := \max_i |\rho(s_i)|$ is much smaller than the number of state variables, i.e. $\mathcal{P} \ll K$. Informally, we say that factored MDPs with smaller ratios $\mathcal{P} : K$ are "more factored."[4]

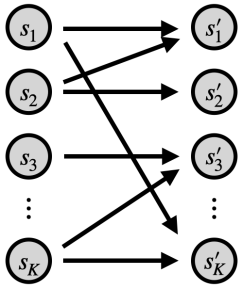


Figure 2.4: Visualization of a factored MDP. In this example, $\rho(s'_1) = \{s_1, s_2\}$, $\rho(s'_2) = \{s_2\}$, etc.

We can visualize a factored MDP as a two-layer Bayesian network (see Figure 2.4), where nodes correspond to state variables at two subsequent time steps, and edges denote parent dependencies. Increased factorization corresponds to a sparser set of edges.

In theory, factored MDPs offer a significant computational advantage over their non-factored counterparts. When agents know the factor-dependencies, they can leverage that structure to learn exponentially faster (Kearns and Koller, 1999; Guestrin et al., 2002). In the general case, methods like SLF-R-MAX (Strehl et al., 2007) and MET-R-MAX (Diuk et al., 2009) can help to learn that structure, although their efficiency relies heavily on having sparse dependencies (low $\mathcal{P}$). See Hester and Stone (2012) (section 6) for a detailed review of these and other factored MDP methods.

Unfortunately, in the general learning setting, agents must not only learn the dependency structure but also the factors themselves. While there has been some recent work on teasing apart these factors of variation (Higgins et al., 2017; Thomas et al., 2017; Sawada et al., 2018), it remains an open question how to best carve up observations into useful internal state variables. For this reason, we will primarily use factored structure in the context of planning. In Chapter 5 we assume the state factors are known and consider action abstractions that increase factorization by intentionally modifying only a small number of variables.

2.1.5 Model-Based RL

Most reinforcement learning algorithms assume the transition and reward functions, T and R, are unknown to the agent. In this setting, the agent must choose between two classes of methods: *model-based* methods, which seek to explicitly estimate the transition and reward functions through interactions with the environment, and *model-free* methods, which simply optimize behavior without them.[5] In model-based RL, once the agent has

[4] It is common to assume the reward function factorizes as well, although we do not require such an assumption here.

[5] For our purposes, this binary is sufficient, although van Hasselt et al. (2019) argue that there is really more of a spectrum, and that methods that use experience replay (Mnih et al., 2015) represent a kind of middle ground.

estimated an internal world model, it can use that model to consider the effects of hypothetical actions.[6] These imagined experiences can exploited in two main ways: first, as additional data to improve the value function or policy via the same methods as model-free RL; and second, as a way to aid action selection by looking ahead at the estimated value of future states. The former is sometimes called "background planning", and the latter is called "decision-time planning" (Sutton and Barto, 2018). However, since the agent is *learning* the world model from data, both of these types of planning are distinct from the planning frameworks we discuss in the next section.

2.2 Planning

In contrast with reinforcement learning, planning frameworks assume the agent has access to an accurate internal world model of its environment. Where this model comes from is not important, and it is perfectly acceptable to suppose it could have been learned through model-based RL. Regardless, the assumption is that querying the internal model with a proposed action is precisely as good as executing the same action in the real environment. The format of the agent's world model depends on which specific planning framework is being used to formalize the decision-making problem, but most planning frameworks assume a factored state space, similar to that of Section 2.1.4.

Since our aim is general-purpose problem solving, we'd like the agent to be able to solve many tasks using the same world model. We could in principle accomplish this with the MDP framework if we used a different reward function for each task; however, most planning formalisms take a different approach. Instead of a reward function, the agent's *goal* is specified in terms of constraints on the state variables (or some subset thereof) within the factored state space. Each planning *task* includes both a goal condition, and an initial state s_0. Meanwhile, the planning *domain* (i.e. the world model) encompasses the transition function, factored state space, and action applicability function, and these quantities are fixed for all tasks. This separation between task and domain allows the agent to use the same world model to solve many different planning problems.

The agent's objective is to find a *plan*, a sequence of actions that connects initial state s_0 to any state s_G that satisfies the specified goal G. We can visualize this process using a *search tree*, as in Figure 2.5.[7] Each node in the tree corresponds to a particular state, and at the beginning of planning, only the initial state is revealed. The agent can then query the world model from any known state s to determine the effect of taking an action a, at which point the agent adds the resulting state s' to the search tree, connecting s to a new node s' via edge a. The process continues until the agent reveals some state s_G that satisfies the goal condition, and the action edges along the path from s_0 to s_G constitute a plan that solves the task.

There are a few ways to measure plan quality and planning efficiency. In some planning paradigms, actions

[6] Note that we are overloading the word *model* here. Previously, we discussed several *problem models* that each formalized the overall decision-making process. Now we are discussing *world models*, functions internal to the agent that produce simulated experiences.

[7] Strictly speaking, we should say *search graph*, since some sequences of actions may return the agent to a previously encountered state, but unless we require optimal plans, the agent can simply ignore these loops.

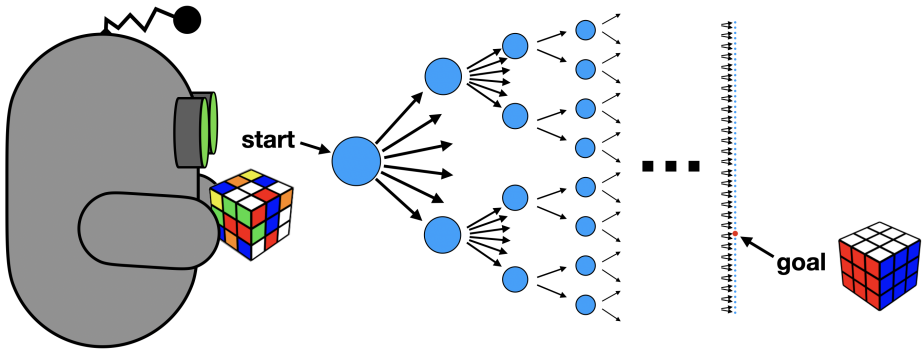


Figure 2.5: Visualizing the planning process as a search tree.

can have associated costs, and an *optimal* plan is one that minimizes the sum of these costs. We might also care specifically about finding a *shortest cost-optimal plan*: a plan with the fewest steps from among the set of lowest-cost plans. Alternatively, we can ignore costs and focus on *satisficing* solutions—that is, finding a plan as quickly as possible, regardless of cost. We can measure efficiency in terms of the number of model queries, or the number of states generated, before finding a plan. For simplicity, we will assume in this dissertation that the transition dynamics are *deterministic*—i.e. actions have exactly one possible effect for each state—which means only one world model query is necessary to determine the effect of an action.

The cost of finding a suitable plan scales exponentially with the length of the shortest possible solution, which usually makes exhaustive exploration of the search tree intractable. *Heuristic search* eases the computational burden by guiding the search towards the most promising states (Bonet and Geffner, 2001). A heuristic function $h(s)$ estimates the number of actions required to reach the goal from a given state s, and typically some variation of "best-first" search then expands states with lower heuristic values first. Some planning frameworks, such as classical planning, are compatible with sophisticated heuristics that exploit structure in the world model to build accurate cost estimates. Other frameworks, like black-box planning, introduce no additional structure whatsoever and have much more limited heuristic options. We will discuss a few of these frameworks, and the differences between them, in the remainder of this section.

2.2.1 Classical Planning

The classical planning formalism imposes structure on a planning problem through objects, predicates, and action schemas. Objects correspond to the various aspects of the environment, and predicates are functions that facilitate the expression of true-or-false propositions about those objects. These propositions are somewhat analagous to the state variables in a factored MDP (see Section 2.1.4), albeit in a different format: typically only *true* propositions are explicitly listed in the state, and any other propositions are assumed to be false. And just like in a factored MDP, the resulting state space can be enormous—even for a rather modest set of objects—since it includes every possible combination of truth values for the predicates. Fortunately, action schemas provide a concise way to specify transition dynamics for the combinatorial number of possible states.

An action schema (sometimes called an *operator*) consists of a *precondition* and an *effect*. The precondition—which is analagous to the action applicability function of a Markov decision process—specifies which predicates need to be true in order to execute the action. The effect—which corresponds to the transition dynamics—specifies which predicates will become either true or false after executing the action. If predicates do not appear in an action's precondition or effect, they are ignored or left unchanged, respectively. We illustrate these concepts in the following example.

Example 2 (Elevator Scheduling). Suppose there are N passengers, distributed among K different floors of a building, who are all waiting for a single elevator. The planning agent's job is to decide on a sequence of elevator controls that will bring every passenger to their desired destination. We'd like to express this problem (a variation of the MICONIC domain first introduced by Koehler and Schuster (2000)) using the framework of classical planning.

The standard way to define a classical planning problem is using PDDL, the Planning Domain Definition Language (McDermott et al., 1998). We first use placeholder objects to define some useful predicates, and then use those predicates to express the elevator controls as action schemas:

```
(define (domain miconic)
  (:predicates:
     (flr ?f) ;; if ?f is a valid floor
     (psgr ?p) ;; if ?p is a valid passenger
     (origin ?p ?f) ;; if passenger ?p originates on floor ?f
     (destin ?p ?f) ;; if passenger ?p wants to reach destination floor ?f
     (served ?p) ;; if passenger ?p has reached their destination
     (in ?p) ;; if passenger ?p is in the elevator
     (at ?f) ;; if the elevator is at floor ?f
     (above ?f1 ?f2) ;; if floor ?f2 is above floor ?f1
  )
  (:action up ;; move the elevator upwards from floor ?f1 to floor ?f2
     :parameters (?f1 ?f2)
     :precondition (and (flr ?f1) (flr ?f2) (at ?f1) (above ?f1 ?f2))
     :effect (and (at ?f2) (not (at ?f1)))
  )
  (:action down ;; move the elevator downwards from floor ?f1 to floor ?f2
     :parameters (?f1 ?f2)
     :precondition (and (flr ?f1) (flr ?f2) (at ?f1) (above ?f2 ?f1))
     :effect (and (at ?f2) (not (at ?f1)))
  )
  (:action board ;; allow passenger ?p to board the elevator at floor ?f
```

```
      :parameters (?f ?p)
      :precondition (and (flr ?f) (psgr ?p) (at ?f) (origin ?p ?f))
      :effect (in ?p)
   )
   (:action depart ;; allow passenger ?p to depart the elevator at floor ?f
      :parameters (?f ?p)
      :precondition (and (flr ?f) (psgr ?p) (at ?f) (destin ?p ?f) (in ?p))
      :effect (and (not (in ?p)) (served ?p))
   )
)
```

Each action schema accepts some number of placeholder parameters. For example, `up` takes a starting floor (`?f1`) and a destination floor (`?f2`). The preconditions describe what needs to be true for the action to be executable, in terms of the various predicates. For example, `up` requires both parameters to be valid floors (`flr ?f1` and `flr ?f2`), requires the elevator to be at the starting floor (`at ?f1`), and requires the destination floor to be above the starting floor (`above ?f1 ?f2`). The effects describe how executing the action will change the relevant predicates. For example, `up` causes the elevator to be at the destination floor (`at ?f2`), and *negates* the original fact that the elevator is at the starting floor (`at ?f1`).

With the above domain in place, we can express a planning problem involving any number of floors and passengers simply by defining a few additional quantities:

```
(define (problem miconic-2psgr-3flr)
   (:domain miconic)
   (:objects
      p0 p1 ;; passengers
      f0 f1 f2 ;; floors
   )
   (:init
      (psgr p0)
      (psgr p1)
      (floor f0)
      (floor f1)
      (floor f2)
      (above f0 f1)
      (above f0 f2)
      (above f1 f2)
      (origin p0 f1)
```

```
      (origin p1 f2)
      (destin p0 f2)
      (destin p1 f0)
      (at f0)
   )
   (:goal
      (and (served p0) (served p1)))
   )
)
```

The problem contains a concrete set of objects and defines an initial state and a goal condition. This particular problem has three floors, `f0`, `f1`, and `f2`, and two passengers, `p0` and `p1`, who start on floors `f1` and `f2` and want to reach floors `f2` and `f0` respectively. The elevator starts on floor `f0`.

What makes action schemas so powerful is that they are defined in a *lifted* way, which means without reference to any particular object. Lifted action schemas use placeholder objects to define the relationships between the various predicates, and the actions are subsequently *grounded* by replacing these placeholders with specific objects. This allows the domain's predicates and actions to be defined abstractly, for all objects, and then instantiated as necessary with specific objects in each new problem file. Using a standardized format like this means that a domain-independent planner, such as Fast Downward (Helmert, 2006), can implement a wide variety of general-purpose planning algorithms and heuristics without any advance knowledge of the specific planning domain or problem it will need to solve.

Additionally, action schemas provide substantial structure that planning heuristics can leverage when estimating the cost of reaching the goal. Some heuristics use the action schemas directly (Bonet et al., 1997; Hoffmann and Nebel, 2001), while others first *translate* into what's called a *finite domain representation* (Sandewall and Rönnquist, 1986; Helmert, 2006), which presents state variables more explicitly than the standard list of true-valued predicates. Heuristics might then ignore or combine the information contained in various state variables in order to estimate costs (Culberson and Schaeffer, 1998; Helmert et al., 2007). For a detailed discussion of these and many other classical planning heuristics, see Helmert and Domshlak (2009).

2.2.2 Numeric Planning

Numeric planning (introduced in PDDL 2.1; see Fox and Long (2003)) extends the classical planning formalism by allowing for continuous state information alongside traditional binary propositions. This behavior is defined via numeric *fluents*, continuous-valued variables that can change over time, and *functions*, which accept a list of zero or more placeholder objects and define a numeric fluent for each unique set of arguments, similar to predicates. The problem file initializes these fluents to their starting values and the actions in the domain file might affect fluents using various expressions (increase, decrease, assign, add, subtract, multiply, divide, etc.)

or check them as part of a precondition (e.g. by testing for equality, greater-than, or less-than relationships).

Numeric planning presents a number of challenges that stem from the interaction between discrete and continuous variables (Dornhege et al., 2012) and that are outside the scope of this dissertation. Typically, classical planning techniques must be adapted before they can work well with numerics (Scala et al., 2016b). However, we will see in Chapter 6 that this isn't always the case; carefully constructed world-model abstractions can improve planning efficiency while handling numeric fluents and binary predicates at the same time.

2.2.3 Black-Box Planning

Planning requires a world model, but most planning research ignores the question of where such a model comes from. In practice, world models tend to come from a system designer who knows PDDL, or in some cases, SAS$^+$, a popular finite domain representation language (Bäckström and Nebel, 1995). Recent work has sought to enable agents to automatically construct symbolic models similar to those designed by human experts (Konidaris et al., 2018; Asai and Fukunaga, 2018), but such methods tend to rely on strong assumptions about the structure of the environment.

Black-box planning (Lipovetzky et al., 2015; Jinnai and Fukunaga, 2017) offers another approach with no assumptions at all. Whereas classical planning provides detailed action schemas that describe abstract preconditions and effects, black-box planning just provides the agent with a simple interface: the agent can query the world model with a hypothetical state to determine which actions would be applicable, or with a state-action pair to determine the resulting state. Black-box planning is appealing because it dramatically lowers the barrier to planning with a learned model. Any world model capable of determining action applicability and simulating action executions is compatible with the black-box planning framework. In fact, just as with model-based reinforcement learning, the model need not even be entirely accurate to be useful.

The downside of black-box planning is that agents are severely limited in terms of heuristic search. Without a formal domain description, there is little information available for heuristics to use when estimating the distance to the goal. One of the few domain-independent heuristics that is compatible with black-box planning is the *goal-count heuristic* (Fikes and Nilsson, 1970), which counts the number of state variables that differ between a given state and the goal. The two basic assumptions of the goal-count heuristic are: a factored state space (i.e. there are state variables to count), and a known goal condition (i.e. there is a reason to count variables). A third, more subtle assumption is that the problem can be decomposed into subproblems, where each state variable can be treated as an approximately independent subgoal. Unfortunately, this subgoal independence assumption is invalid for most planning problems of practical interest, and thus the goal-count heuristic is often misleading. In Chapter 5 we will look at ways to improve the accuracy of the goal-count heuristic through action abstraction while maintaining the desirable generality of the black-box formalism.

2.2.4 Open-Scope Planning

Modern AI planning is both extremely general-purpose and profoundly inflexible. The promise of domain-independent planners is that a single program can be used to solve planning tasks arising from many specific applications. This promise has largely been realized: given an appropriately specified model of a planning problem, modern planners can quickly tackle anything from game playing (Korf, 1985a; Korf, 1985b) to transportation logistics (Refanidis et al., 2001) to chemical synthesis (Matloob and Soutchanski, 2016). But these impressive achievements rely on the fundamental assumption that the domain model is always well-matched to the task. Each application uses a different domain model that was carefully designed by human experts, and while each domain supports multiple problems, the problems within a given domain all share significant structure. This significantly limits generality, because a planner cannot solve logistics problems with a chemical synthesis model. In other words, the models have limited *scope*.

A general-purpose agent must go beyond the limitations of current domain-independent planners by acquiring and maintaining an *open-scope* model—one rich enough to describe any planning task it may encounter during its deployment. Such an open-scope model will necessarily contain large amounts of information irrelevant to any individual task, because the agent can no longer rely on human experts to provide models for new domains (Konidaris, 2019). Unfortunately, when learned models contain large amounts of irrelevant information, the search space grows exponentially and planning quickly becomes intractable. Recent work has shown that many state-of-the-art planners suffer significant reductions in performance when irrelevant objects, state variables, or operators are included in domain descriptions (Vallati and Chrpa, 2019; Silver et al., 2021). We address this problem in Chapter 6 by using task-specific structure to learn world-model abstractions that simplify open-scope models to the point where planning once again becomes tractable.

2.3 Abstraction

To succeed at learning and planning, agents must construct abstractions that allow them to express each decision problem at an appropriate level of detail. For high-level tasks like playing chess, cleaning a home, or driving a car, reasoning at the level of individual pixels and motor torques is clearly hopeless. Agents can use abstraction to reduce complexity by discarding certain irrelevant information, but determining which information is relevant can be challenging, and discarding too much can impede decision making. Abstraction therefore introduces a tradeoff between richness and simplicity.

Formalizing the concept of abstraction will require us to be specific about which information the agent will ignore. In principle, any aspect of the formal decision-making models introduced earlier in this chapter would be a reasonable target for abstraction. Here we will restrict our focus to abstractions of observations, histories, actions, and world models.

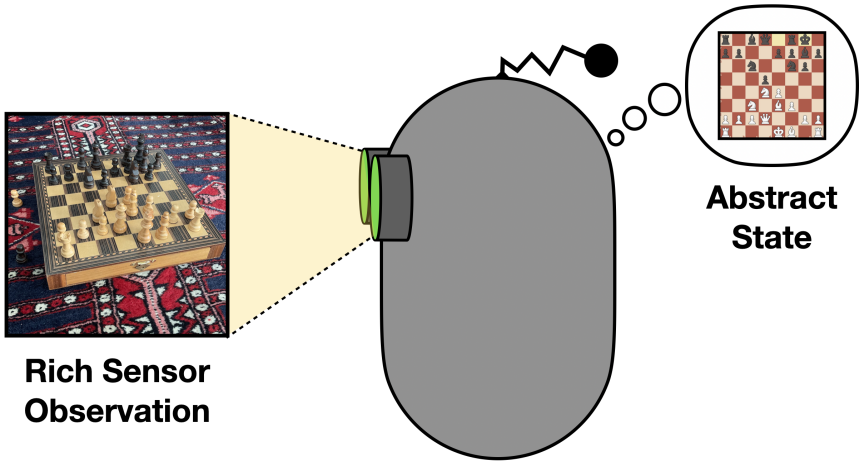


Figure 2.6: Observation abstraction.

2.3.1 Observation Abstractions

The first place we will apply abstraction is on the agent's input observations. To avoid challenges arising from partial observability, let us adopt the block MDP model for now,[8] where system behavior is governed by some underlying set of hidden states $\mathcal{S}$, but where the agent's sensors σ produce observations $s \overset{\sigma}{\mapsto} \omega$ belonging to the larger set Ω. To support decision making when Ω is too noisy or high-dimensional for tractable learning, we'll turn to observation abstraction. Of course, the block MDP assumption means that observations are themselves Markov states, so in this context, it is equivalent to talk about *state* abstraction.

Our objective is to find an abstraction function $\phi : \Omega \to Z$ mapping each observation ω to an abstract state $z = \phi(\omega)$, with the hope that learning is tractable using the abstract representation Z (see Figure 2.6). We often refer to $\omega \in \Omega$ as *ground* observations (or ground states), and M as the ground MDP, to reflect that these quantities are *grounded*, as opposed to abstract, i.e. they have a firm basis in the true environment.[9] Since our goal is to support effective abstract decision making, we are mainly concerned with the policy class Π_ϕ, the set of policies with the same behavior for all observations that share the same abstract state:

$$\Pi_\phi := \left\{\pi : \big(\phi(\omega_1) = \phi(\omega_2)\big) \implies \big(\pi(a|\omega_1) = \pi(a|\omega_2)\big),\ \forall\, a \in A; \omega_1, \omega_2 \in \Omega\right\}. \tag{2.3}$$

An abstraction $\phi : \Omega \to Z$, when applied to an MDP M, induces a new abstract decision process $M_\phi = (Z, A, T^\pi_{\phi,t}, R^\pi_{\phi,t}, \gamma)$, whose dynamics may depend on the current timestep t or the agent's behavior policy π, and, crucially, which is not necessarily Markov.

Consider the following example.

[8] We will subsequently address partial observability through history abstractions.

[9] This use of the term "grounded" has a similar connotation to that of a grounded action schema in classical planning. In both cases, we are distinguishing something concrete from something abstract.

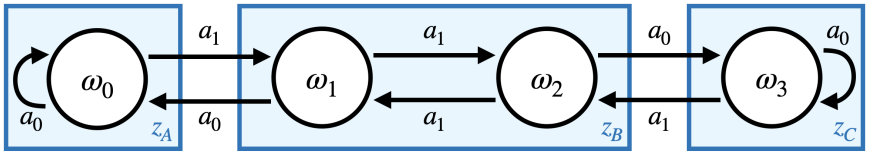


Figure 2.7: An MDP and a non-Markov abstraction.

The figure above depicts an MDP with two distinct actions (arrows) and four states/observations (white circles). It also depicts an abstraction function ϕ (blue rectangles) that aggregates ω_1 and ω_2 into a single abstract state z_B. It's common to view state abstraction as aggregating or partitioning ground states into abstract states in this way (Li et al., 2006).

State aggregation usually involves choosing a fixed weighting scheme $W(\omega)$ to express how much each ground state ω contributes to its abstract state z, where the weights sum to 1 for the set of ω in each z. We can then define the abstract transition dynamics T_ϕ as a W-weighted sum of the ground dynamics (and similarly for reward): $T_\phi(z'|a,z) = \sum_{\omega' \in z'} \sum_{\omega \in z} T(x'|a,x)w(x)$. A natural choice for $W(\omega)$ is to use the ground-state visitation frequencies. For example, if the agent selects actions uniformly at random, this leads to all observations being equally likely, and so $W(\omega_0) = W(\omega_3) = 1$ and $W(\omega_1) = w(\omega_2) = 0.5$.

In this formulation, the abstract decision process is assumed to be Markov by construction, and T_ϕ and R_ϕ are assumed not to depend on the policy or timestep. But this is an oversimplification. The abstract transition dynamics are *not* Markov; they change depending on how much history is conditioned on: $T_\phi(z_A|a_t = a_0, z_t = z_B) = 0.5$, whereas $\Pr(z_A|a_t = a_0, z_t = z_B, a_{t-1} = a_1, z_{t-1} = z_A) = 1$. By contrast, the ground MDP's dynamics are fully deterministic (and Markov). Clearly, if we define the abstract MDP in this way, it may not match the behavior of the original MDP.[10] Even worse, if the agent's policy changes—such as during learning—this discrepancy can cause RL algorithms with bounded sample complexity in the ground MDP to make an arbitrarily large number of mistakes in the abstract MDP (Abel et al., 2018).

For the abstract decision process to faithfully simulate the ground MDP's dynamics, $W(\omega)$ must be allowed to vary such that it always reflects the correct ground-state frequencies. Unfortunately, even in the simple example above, maintaining accurate weights for $W(\omega)$ requires keeping track of an unbounded amount of history: if an agent reaches abstract state z_B and repeats action a_1 an arbitrarily large number of times (N), then knowing precisely which ground state it will end up in (ω_1 or ω_2) requires remembering the abstract state it originally came from $N+1$ steps prior. Successful modeling of the transition dynamics for a subsequent action a_0 hinges on exactly this distinction between ω_1 and ω_2. The abstraction introduces partial observability, and to compensate, $W(\omega)$ must be replaced with a belief distribution over ground states, conditioned on the entire history; instead of an abstract MDP, we have an abstract POMDP (Bai et al., 2016). This is especially unsatisfying, not just because learning is challenging in POMDPs (Zhang et al., 2012), but because the block MDP formulation was supposed to avoid precisely this type of partial observability: an abstraction exists (namely $\phi = \sigma^{-1}$) that would result in a fully observable abstract MDP, if only the agent knew what it was.

[10] Abel et al. (2018) presented a three-state chain MDP where they made a similar observation.

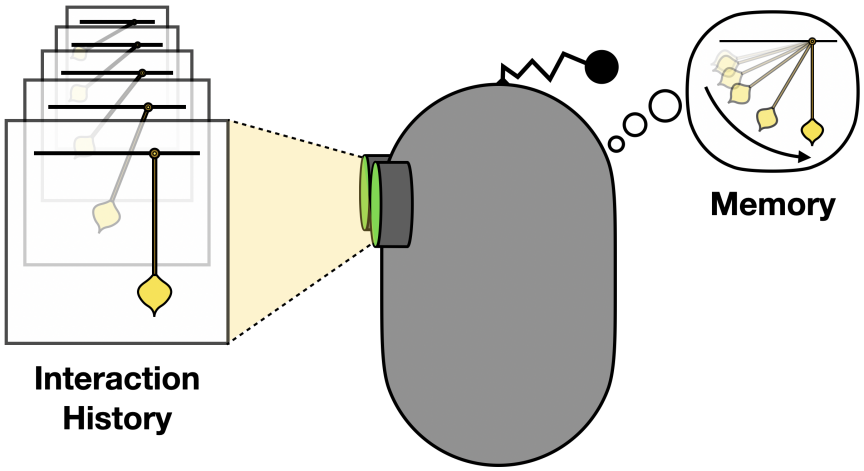


Figure 2.8: History abstraction.

In Chapter 3 we will see that it's possible to learn a state abstraction that both reflects the behavior of the underlying ground MDP *and* preserves the Markov property in the abstract MDP, without ever estimating or maintaining a belief distribution. Our theoretical results leverage inverse dynamics models and contrastive learning to derive conditions under which a state abstraction is Markov. We then adapt these conditions into a corresponding training objective for learning abstractions directly from experiences in the ground MDP.

2.3.2 History Abstractions

We will use history abstractions to aid decision making in non-Markov decision processes. As noted earlier, making optimal decisions in partially observable environments may require the agent to condition its behavior on observation histories of unbounded and variable length. History abstractions offer a less cumbersome alternative.

We can view history abstractions in two seemingly opposite ways. In one sense, history abstractions simply apply the same principle as observation abstractions, but generalize the input space from a single observation to an observation sequence. In another sense, history abstractions act like the *inverse* of observation abstractions: rather than distilling an overly complicated observation space into a more parsimonious one, history abstractions instead *augment* an impoverished observation space to make it richer. This view constitutes a form of feature augmentation (see Section 2.1.2), though it need not involve a human expert. These two views are mutually compatible, and we will alternate between them based on whichever offers the clearest explanation.

Formally, we define a history abstraction μ as a *memory function* mapping from an interaction history $h_t = (a_t, \omega_t, a_{t-1}, \omega_{t-1}, \dots)$ to a memory state, $h_t \overset{\mu}{\mapsto} m_{t+1}$, within a set of possible memory states $\mathcal{M}$, that we use to augment the agent's next observation ω_{t+1}. The agent's behavior is conditioned on this memory-augmented observation (ω_{t+1}, m_{t+1}), rather than on the entire history h_{t+1} or the observation ω_{t+1} alone. By restricting memories to have a fixed size, we thus avoid problems arising from unbounded and variable-length

histories while still facilitating better decisions.

This model of history abstractions is compatible with simple stateless memory functions like frame-stacking, stateful functions that perform incremental updates to existing memories, history aggregation akin to the aggregations of states discussed in the previous section, and neural networks that implement arbitrarily complicated versions of either. Memory functions combine the most promising aspects of other approaches to generalize reinforcement learning to non-Markov decision processes. Like predictive state representations, they are functions only of observable quantities; like feature augmentation and belief-state approaches, they are functions of the agent's past, not predictions about possible futures. We visualize a simple 2-state memory function in Figure 4.1b that supports learning an optimal policy for the problems in Figure 4.1a.

But how should an agent learn such a memory function in order to maximize value? How are memory functions and value functions related? In Chapter 4 we will propose one useful way of connecting these concepts. We will use the structure of value functions to both detect when a decision process is non-Markov and subsequently restore the Markov property using memory. We also prove that in so doing, the optimal policy will then be expressible solely in terms of the memory-augmented observations.

2.3.3 Action Abstractions

Useful abstractions must ultimately simplify either the agent's inputs or its outputs. Observation and history abstractions deal with inputs; they simplify decision making by either obfuscating or revealing differences between between the agent's observations. Action abstractions deal with outputs; they directly restrict the set of possible decisions under the agent's consideration.

Action abstractions can operate within two distinct time regimes: instantaneously, or over some extended duration. Instantaneous action abstraction simplifies decision making at a single timestep by ruling out unpromising or duplicate actions. Extended action abstraction limits the *number* of timesteps that require decisions by introducing high-level skills that run until they accomplish some useful objective. The latter is generally more common and is what we will focus on here; however, it is possible to view the world-model abstractions in the next section as an example of the former.

There are many ways to formalize the notion of skills for learning and planning. For a detailed look at skills in the learning setting, see: Sutton et al.'s (1999) framework defining *options*, or time-extended actions, consisting of preconditions, specialized policies, and termination conditions; and Dayan and Hinton's (1992) *feudal reinforcement learning*, which constructs sub-agents to which the agent can delegate some of its decisions. Here we will mainly concern ourselves with the planning setting—and black-box planning, in particular—in which we will formalize skills as follows.

We define a planning skill as a *macro-action* (or simply *macro*), a deterministic sequence of actions,[11] typically

[11] For environments with probabilistic effects, macros could in principle be generalized to more complex abstract skills incorporating state information, but that extension is more closely related to the reinforcement learning setting and is beyond the scope of this dissertation.

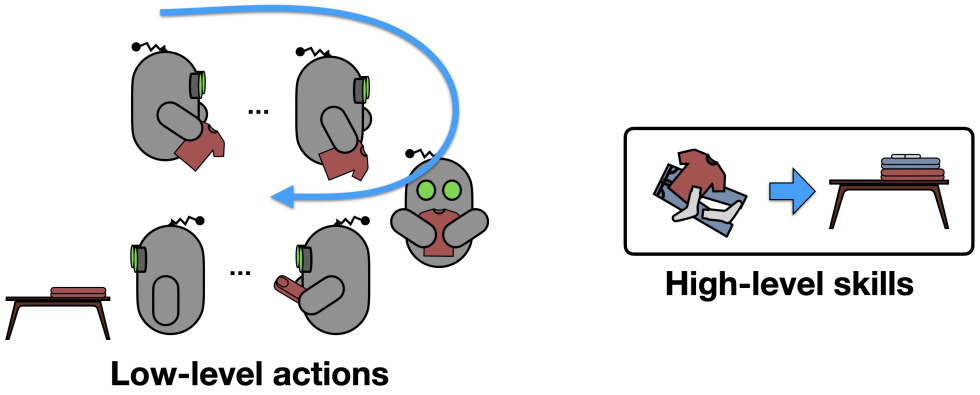


Figure 2.9: Action abstraction.

for the purpose of accomplishing some useful subgoal. To avoid confusion, we often refer to the original non-macro actions as "primitive" actions. Macro-actions have parameters, preconditions, and effects, just like primitive actions, though in black-box planning, we again assume that the planner does not have access to such a declarative description. Instead, when planning with black-box macro-actions, there are two alternatives. Either the world model must be updated to additionally compute *macro-action* validity and effects in a single step, or alternatively, each primitive action in the macro must be simulated sequentially, with longer macros requiring more world-model queries.

When macro-actions are added to the set of primitive actions, they can reduce search tree depth at the expense of increasing the branching factor. In some cases, this has been shown to improve planning efficiency, particularly when the macro-actions cause the problem's subgoals to become independent (Korf, 1985b). In Chapter 5, we further explore this idea in the context of black-box planning by introducing macros that reflect the problem's factored structure and hence align more closely with the subgoal-independence assumption of the goal-count heuristic.

2.3.4 World-Model Abstractions

As general-purpose agents spend time in the world, their internal world model will ideally grow to encompass the various tasks they've encountered. After a while, such a world model may become quite detailed and contain hundreds of skills useful for accomplishing a wide variety of objectives. But the greater the capability of the agent's world model, the greater the cost of planning within it, since new actions increase the branching factor of the search tree. In the previous section, we defined action abstractions that essentially amounted to shortcuts through the search tree. Now we will consider abstractions that remove entire branches.

The vast majority of tasks the agent will face require only a small fraction of its (open-scope) world model. This intuition naturally leads to the concept of world-model abstractions that ignore or remove from consideration all those aspects of the agent's world model not relevant to the task at hand. Given a grounded classical or numeric planning problem, we formally define a world-model abstraction as a simplification of the planning

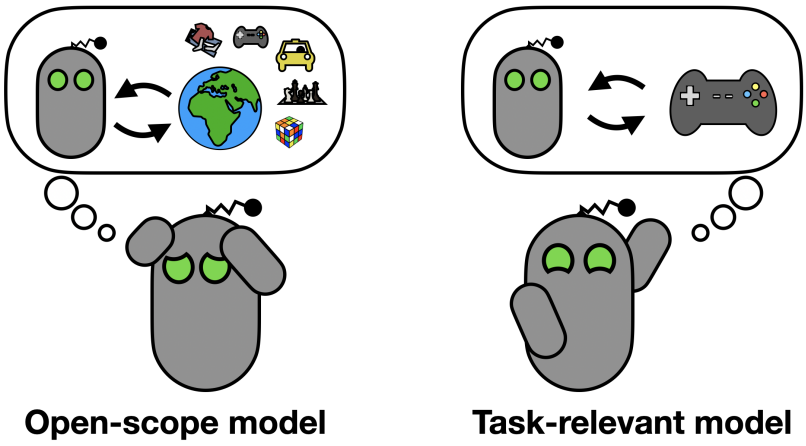


Figure 2.10: World-model abstraction.

problem that deletes certain actions and/or objects prior to planning. This is essentially a form of instantaneous action abstraction (see Section 2.3.3), since it prevents the planner from selecting actions that were deleted or that involve deleted objects. As with other forms of abstraction, the challenge is to simplify decision making without degrading solution quality. In Chapter 6 we explore several ways to exploit the structure of the agent's task to build world-model abstractions that delete irrelevant information and make planning more efficient while provably preserving optimal plans.

Chapter 3

Observation Abstractions

> We make sense of the world intentionally. Faced with chaos, we seek or make the familiar, and build up the world with it. Babies do it, we all do it; we filter out most of what our senses report.
>
> —Ursula K. Le Guin

Reinforcement learning in Markov decision processes with rich observations requires a suitable state representation. Such representations can sometimes emerge as a byproduct of simply doing reinforcement learning with deep neural networks. However, in domains where precise and succinct expert state information is available, agents trained on such expert state features usually outperform agents trained on rich observations. Recent work has sought to close this *representation gap* by incorporating a wide range of representation-learning objectives that help the agent learn abstract state representations with various desirable properties.

The fundamental structural assumption of the Markov decision process model, and thus the most obvious property to incentivize in a state representation, is the Markov property. Recall that the Markov property holds if and only if the state representation contains enough information to accurately characterize the rewards and transition dynamics of the decision process. Markov decision processes have this property by definition, and most reinforcement learning algorithms depend on having Markov state representations. For instance, the ubiquitous objective of achieving an optimal policy that depends only on the current state is impossible to guarantee without *Markov* states.

But, as we saw earlier, learned abstract state representations are not necessarily Markov, even when built on top of MDPs. This is due to the fact that abstraction necessarily throws away information. Discard too much information, and the resulting representation cannot accurately characterize the environment. Discard too little, and agents will fail to close the representation gap. Abstraction must balance between ignoring irrelevant information and preserving what is important for decision making.

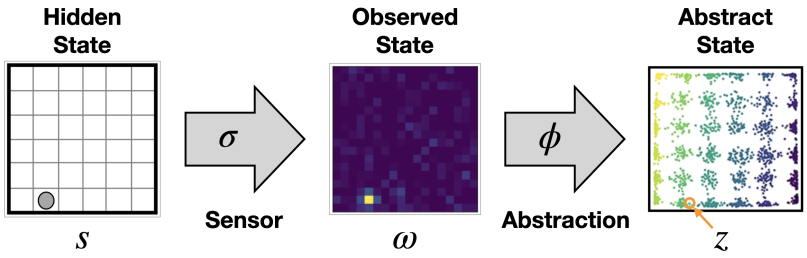


Figure 3.1: A 6×6 visual gridworld domain with hidden state s and unknown sensor σ, where an abstraction function ϕ maps each high-dimensional observed state ω to a lower-dimensional abstract state z (orange circle).

There are several ways to incentivize learning a Markov state representation. If reward feedback is available, an agent can use it to determine which state information is relevant to the task at hand. Alternatively, if the agent can predict ground observations from learned abstract states, then its abstraction preserves *all* available information, and the Markov property comes along for free. (This assumes ground observations are already Markov, like in a block MDP; we will consider non-Markov observations in the next chapter.) However, these approaches are impractical if rewards are infrequent or non-existent, or observations are sufficiently complex.

In this chapter we will explore a new approach to learning Markov state abstractions. We begin by defining a set of theoretical conditions that are sufficient for an abstraction to retain the Markov property. We next show that these conditions are approximately satisfied by simultaneously training an inverse model to predict the action distribution that explains two consecutive states, and a discriminator to determine whether two given states were in fact consecutive. Our combined training objective (architecture shown in Fig. 3.2) supports learning Markov abstract representations without requiring reward information or observation prediction.

Our method is effective for learning Markov state abstractions that are highly beneficial for decision making. We perform evaluations in two settings with rich, visual observations: a gridworld navigation task (Fig. 3.1) and a set of continuous control benchmarks. In the gridworld, we construct an abstract representation offline—without access to reward feedback—that captures the underlying structure of the domain and fully closes the representation gap between visual and expert features. In the control benchmarks, we train the abstraction objective concurrently with (online) reinforcement learning, where it leads to a significant performance improvement over state-of-the-art visual representation learning.

3.1 Related Work

3.1.1 Bisimulation

The idea of learning Markov state abstractions is related to the concept of bisimulation (Dean and Givan, 1997), the strictest type of state aggregation discussed in Li et al. (2006), where ground states are equivalent if they have exactly the same expected reward and transition dynamics. Preserving the Markov property is a prerequisite for a bisimulation abstraction, since the abstraction must also preserve the (Markov) ground-state

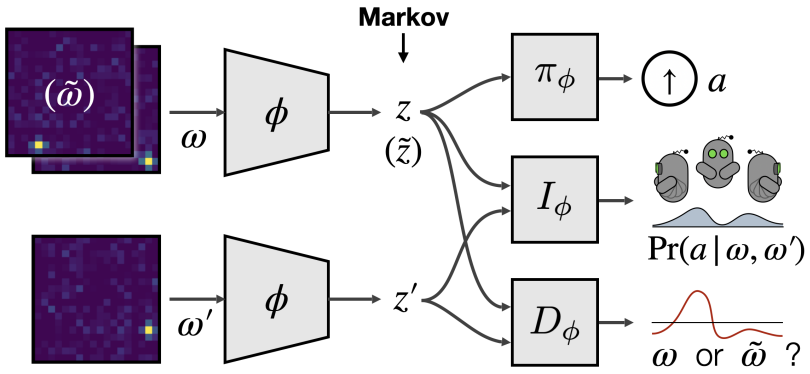


Figure 3.2: Markov abstraction training architecture. A shared encoder ϕ maps ground states ω, ω' to abstract states z, z', which are inputs to an inverse dynamics model I and a contrastive model D that discriminates between real ($\omega \to \omega'$) and fake ($\tilde{\omega} \to \omega'$) state transitions. The agent's policy π depends only on the current abstract state.

transition dynamics. Bisimulation-based abstraction is appealing because, by definition, it leads to high-fidelity representations. But bisimulation is also very restrictive, because it requires an abstraction to be Markov for *any* policy, even for policies that can't be expressed under the abstraction (i.e. policies not in Π_ϕ).

Subsequent work on approximate MDP homomorphisms (Ravindran and Barto, 2004) and bisimulation metrics (Ferns et al., 2004; Ferns et al., 2011) relaxed these strict assumptions and allowed ground states to have varying degrees of "bisimilarity." Castro (2020) introduced a further relaxation, π-bisimulation, which measures the behavioral similarity of states *under a policy* π. But whereas full bisimulation can be too strong, since it constrains the representation based on policies the agent may never actually select, π-bisimulation can be too weak, since if the policy deviates from π (e.g. during learning), the metric must be updated, and the representation along with it. Our approach can be thought of as a useful compromise between these two extremes.

While bisimulation-based approaches have historically been computationally expensive and difficult to scale, recent work has started to change that (Castro, 2020; Lehnert and Littman, 2020; van der Pol et al., 2020; Biza et al., 2021). Two recent algorithms in particular, DeepMDP (Gelada et al., 2019) and Deep Bisimulation for Control (DBC) (Zhang et al., 2021), learn approximate bisimulation abstractions by training the abstraction end-to-end with an abstract transition model and reward function. This is a rather straightforward way to learn Markov abstract state representations since it effectively encodes Definition 1 as a loss function.

One drawback of bisimulation-based methods is that learning an accurate model can be challenging and typically requires restrictive modeling assumptions, such as deterministic, linear, or Gaussian transition dynamics. Bisimulation methods may also struggle if rewards are infrequent or if the abstraction must be learned without access to rewards. Jointly training an abstraction ϕ with only the transition model $\hat{T}(\phi(x), a) \approx \phi(x')$ can easily lead to a trivial abstraction like $\phi(x) \mapsto 0$ for all x, since ϕ produces both the inputs and outputs for the model. Our approach to learning a Markov state abstraction avoids this type of representation collapse without learning a forward model, and is less restrictive than bisimulation, since it is compatible with both reward-free and reward-informed settings.

3.1.2 Ground-State Prediction and Reconstruction

Ground-state (or pixel) prediction (Watter et al., 2015; Song et al., 2016; Kaiser et al., 2020) mitigates representation collapse by forcing the abstract state to be sufficient not just for predicting future *abstract* states, but also future *ground states*. Unfortunately, in stochastic domains, this comes with the challenging task of estimating probability distributions over the ground state space, and as a result, performance is about on-par with end-to-end deep RL (van Hasselt et al., 2019). Moreover, both pixel prediction and the related task of pixel reconstruction (Mattner et al., 2012; Finn et al., 2016; Higgins et al., 2017; Corneil et al., 2018; Ha and Schmidhuber, 2018; Yarats et al., 2019; Hafner et al., 2020; Lee et al., 2020) are misaligned with the fundamental goal of state abstraction. These approaches train models to perfectly reproduce the relevant ground state, ergo the abstract state must effectively throw away no information. By contrast, the objective of state abstraction is to throw away *as much information as possible*, while preserving only what is necessary for decision making. Provided the abstraction is Markov and accurately simulates the ground MDP, we can safely discard the rest of the observation.

3.1.3 Inverse Dynamics Models

As an alternative to (or in addition to) learning a traditional world model that predicts forward in time, it is sometimes beneficial to learn an inverse model. An inverse dynamics model $I(a|\omega', \omega)$ instead predicts the distribution over actions that could have resulted in a transition between a given pair of states, essentially explaining *why* the state transition occurred. Inverse models have been used for improving generalization from simulation to real-world problems (Christiano et al., 2016), enabling effective robot motion planning (Agrawal et al., 2016), defining intrinsic reward bonuses for exploration (Pathak et al., 2017; Choi et al., 2019), and decoupling representation learning from rewards (Zhang et al., 2018). But while inverse models often help with representation learning, we show in Sec. 3.2.2 that they are insufficient for ensuring a Markov abstraction.

3.1.4 Contrastive Learning

Since the main barrier to effective next-state prediction is learning an accurate forward model, a compelling alternative is contrastive learning (Gutmann and Hyvärinen, 2010), which sidesteps the prediction problem and instead simply aims to decide whether a particular state, or sequence of states, came from one distribution or another. Contrastive loss objectives typically try to distinguish either sequential states from non-sequential ones (Shelhamer et al., 2016; Anand et al., 2019; Stooke et al., 2020), real states from predicted ones (van den Oord et al., 2018), or determine whether two augmented views came from the same or different observations (Laskin et al., 2020b). Contrastive methods learn representations that in some cases lead to empirically substantial improvements in learning performance, but none has explicitly addressed the question of whether the resulting state abstractions actually preserve the Markov property. We are the first to show that without forward model estimation, pixel prediction/reconstruction, or dependence on reward, the specific combination

of inverse model estimation and contrastive learning that we introduce in Section 3.2 is sufficient to learn a Markov state abstraction.

3.1.5 Kinematic Inseparability

One contrastive approach which turns out to be closely related to Markov state abstraction is Misra et al.'s (2020) HOMER algorithm, and the corresponding notion of *kinematic inseparability* (KI) abstractions. Two observations ω'_1 and ω'_2 in a block MDP are defined to be kinematically inseparable if $\Pr(\omega, a|\omega'_1) = \Pr(\omega, a|\omega'_2)$ and $T(\omega''|a, \omega'_1) = T(\omega''|a, \omega'_2)$ (which the authors call "backwards" and "forwards" KI, respectively). The idea behind KI abstractions is that unless two observations can be distinguished from each other—by either their backward or forward dynamics—they ought to be treated as the same abstract state. The KI conditions are slightly stronger than the ones we describe in Section 3.2, although when we convert our conditions into a training objective in Section 3.3, we additionally satisfy a novel form of the KI conditions, which helps to prevent representation collapse. While our approach works for both continuous and discrete state spaces, HOMER was only designed for discrete abstract states, and requires specifying—in advance—an upper bound on the *number* of abstract states (which is impossible for continuous state spaces), as well as learning a "policy cover" to reach each of those abstract states (which remains impractical even under discretization). For a more detailed discussion about KI and Markov abstractions, see Appendix A.6.

3.1.6 Other Approaches

The Markov property is just one of many potentially desirable properties that a representation might have. Not all Markov representations are equally beneficial for learning; otherwise, simply training an RL agent end-to-end on (frame-stacked) image inputs ought to be sufficient, and none of the methods in this section would need to do representation learning at all.

Smoothness is another desirable property and its benefits in reinforcement learning are well known (Pazis and Parr, 2013; Pirotta et al., 2015; Asadi et al., 2018). Both DeepMDP (Gelada et al., 2019) and DBC (Zhang et al., 2021), which we compare against, utilize Lipschitz smoothness—an upper bound on how quickly a function can change—when learning abstract state representations. We find in Section 3.5 that a simple smoothness objective helps our approach in a similar way. A full investigation of other representation-learning properties (e.g. value preservation (Abel et al., 2016), symbol construction (Konidaris et al., 2018), suitability for planning (Kurutach et al., 2018), information compression (Abel et al., 2019)) is beyond the scope of this dissertation.

Since our approach does not require any reward information and is agnostic as to the underlying RL algorithm, it would naturally complement exploration methods designed for sparse reward problems (Pathak et al., 2017; Burda et al., 2018). Exploration helps to ensure that the experiences used to learn the abstraction cover as much of the ground MDP's state space as possible. In algorithms like HOMER (above) and the more recent

Proto-RL (Yarats et al., 2021), the exploration and representation learning objectives are intertwined, whereas our approach is, in principle, compatible with any exploration algorithm. Here we focus solely on the problem of learning Markov state abstractions and view exploration as an exciting direction for future work.

3.2 Markov State Abstractions

Recall that for a state representation to be Markov (whether ground or abstract), it must be a sufficient statistic for predicting the next state and expected reward, for any action the agent selects. The state representation of the ground MDP is Markov by definition, but learned state abstractions typically have no such guarantees. In this section, we introduce conditions that provide the missing guarantees.

Accurate abstract modeling the ground MDP requires replacing the fixed weighting scheme $W(\omega)$ of Section 2.3.1 with a belief distribution, denoted by $B_\phi(\omega|\{\cdots\})$, that measures the probability of each ground state ω, conditioned on the entire history of agent experiences. Our objective is to find an abstraction ϕ such that any amount of history can be summarized with a single abstract state z.

When limited to the most recent abstract state z, B_ϕ may be policy-dependent and non-stationary:[1]

$$B^\pi_{\phi,t}(\omega|z) := \frac{\mathbb{1}[\phi(\omega) = z]\, P^\pi_t(\omega)}{\sum_{\tilde{\omega}\in z} P^\pi_t(\tilde{\omega})}, \qquad P^\pi_t(\omega) := \sum_{a\in A}\sum_{\tilde{\omega}\in\Omega} T(\omega|a,\tilde{\omega})\pi_{t-1}(a|\tilde{\omega})P^\pi_{t-1}(\tilde{\omega}), \tag{3.1}$$

for $t \geq 1$, where $\mathbb{1}[\cdot]$ denotes the indicator function, π is the agent's (possibly non-stationary) behavior policy, and P_0 is an arbitrary initial state distribution. Note that P^π_t and $B^\pi_{\phi,t}$ may still be non-stationary even if π is stationary.[2]

We generalize to a k-step belief distribution (for $k \geq 1$) by conditioning (3.1) on additional history:

$$B^{\pi(k)}_{\phi,t}\left(\omega_t|z_t,\{a_{t-i},z_{t-i}\}_{i=1}^k\right) := \\ \frac{\mathbb{1}[\phi(\omega_t) = z_t]\ \sum_{\omega_{t-1}\in\Omega} T(\omega_t|a_{t-1},\omega_{t-1})\, B^{\pi(k-1)}_{\phi,t}\left(\omega_{t-1}\mid z_{t-1},\{a_{t-i},z_{t-i}\}_{i=2}^k\right)}{\sum_{\tilde{\omega}_t\in z_t}\sum_{\tilde{\omega}_{t-1}\in z_{t-1}} T(\tilde{\omega}_t|a_{t-1},\tilde{\omega}_{t-1})\, B^{\pi(k-1)}_{\phi,t}\left(\tilde{\omega}_{t-1}\mid z_{t-1},\{a_{t-i},z_{t-i}\}_{i=2}^k\right)}, \tag{3.2}$$

where $B^{\pi(0)}_{\phi,t} := B^\pi_{\phi,t}$. Any abstraction induces a belief distribution, but the latter is only independent of history for a *Markov* abstraction. We formalize this concept with the following definition.

Definition 2 (Markov State Abstraction). *Given an MDP $M = (\Omega, A, R, T, \gamma)$, initial state distribution P_0, and policy class Π_C, a state abstraction $\phi : \Omega \to Z$ is* Markov *if and only if for any policy $\pi \in \Pi_C$, ϕ induces a belief distribution B^π_ϕ such that for all $\omega \in \Omega$, $z \in Z$, $a \in A$, and $k \geq 1$: $B^{\pi(k)}_{\phi,t}\left(\omega|z_t,\{a_{t-i},z_{t-i}\}_{i=1}^k\right) = B^\pi_{\phi,t}\left(\omega|z_t\right)$.*

[1] This section closely follows Hutter (2016), except here we consider belief distributions over ground states, rather than full histories. An advantage of Hutter's work is that it also considers *abstractions* over histories, though it only provides a rough sketch of how to learn such abstractions. We will take a more detailed look at how to learn history abstractions in Chapter 4.

[2] This can happen, for example, when the policy induces either a Markov chain that does not have a stationary distribution, or one whose stationary distribution is different from P_0.

In other words, Markov state abstractions induce belief distributions that only depend on the most recent abstract state. This property allows an agent to avoid belief distributions entirely, and base its decisions solely on abstract states. Note that Definition 2 is stricter than Markov state representations (Def. 1). An abstraction that collapses every ground state to a single abstract state still produces a Markov state representation, but for non-trivial ground MDPs it also induces a history-dependent belief distribution.

Given these definitions, we can define the abstract transitions and rewards for the policy class Π_ϕ (see Eqn. (2.3)) as follows:[3]

$$T^\pi_{\phi,t}(z'|a,z) = \sum_{\omega' \in z'} \sum_{\omega \in z} T(\omega'|a,\omega) B^\pi_{\phi,t}(\omega|z), \tag{3.3}$$

$$R^\pi_{\phi,t}(z',a,z) = \sum_{\omega' \in z'} \sum_{\omega \in z} \frac{R(\omega',a,\omega)T(\omega'|a,\omega)}{T^\pi_{\phi,t}(z'|a,z)} B^\pi_{\phi,t}(\omega|z). \tag{3.4}$$

Conditioning the belief distribution on additional history yields k-step versions compatible with Definition 1. In the special case where $B_\phi(\omega|z)$ is stationary and policy-independent (and if rewards are defined over state-action pairs), we recover the fixed weighting function $W(\omega)$ of Li et al. (2006).

3.2.1 Sufficient Conditions for a Markov Abstraction

The strictly necessary conditions for ensuring an abstraction ϕ is Markov over its policy class Π_ϕ depend on T and R, which are typically unknown and hard to estimate due to Ω's high-dimensionality. However, we can still find sufficient conditions without explicitly knowing T and R. To do this, we require that two quantities are equivalent in M and M_ϕ: the inverse dynamics model, and a density ratio that we define below. The inverse dynamics model $I^\pi_t(a|\omega',\omega)$ is defined in terms of the transition function $T(\omega'|a,\omega)$ and expected next-state dynamics $P^\pi_t(\omega'|\omega)$ via Bayes' theorem: $I^\pi_t(a|\omega',\omega) := \frac{T(\omega'|a,\omega)\pi_t(a|\omega)}{P^\pi_t(\omega'|\omega)}$, where $P^\pi_t(\omega'|\omega) = \sum_{\tilde{a}\in A} T(\omega'|\tilde{a},\omega)\pi_t(\tilde{a}|\omega)$. The same is true of their abstract counterparts, $I^\pi_{\phi,t}(a|z',z)$ and $P^\pi_{\phi,t}(z'|z)$.

Theorem 1. *If $\phi : \Omega \to Z$ is a state abstraction of MDP $M = (\Omega, A, R, T, \gamma)$ such that for any policy π in the policy class Π_ϕ, the following conditions hold for every timestep t:*

1. ***Inverse Model.*** *The ground and abstract inverse models are equal: $I^\pi_{\phi,t}(a|z',z) = I^\pi_t(a|\omega',\omega)$, for all $a \in A$; $z, z' \in Z$; $\omega, \omega' \in \Omega$, such that $\phi(\omega) = z$ and $\phi(\omega') = z'$.*

2. ***Density Ratio.*** *The ground and abstract next-state density ratios are equal, when conditioned on the same abstract state: $\frac{P^\pi_t(z'|z)}{P^\pi_t(z')} = \frac{P^\pi_t(\omega'|z)}{P^\pi_t(\omega')}$, for all $z, z' \in Z$; $\omega' \in \Omega$, such that $\phi(\omega') = z'$, where $P^\pi_t(\omega'|z) = \sum_{\tilde{\omega}\in\Omega} P^\pi_t(\omega'|\tilde{\omega}) B^\pi_{\phi,t}(\tilde{\omega}|z)$, and $P^\pi_t(z') = \sum_{\tilde{\omega}'\in\Omega} P^\pi_t(\tilde{\omega}') B^\pi_{\phi,t}(\tilde{\omega}'|z')$.*

Then ϕ is a Markov state abstraction.

[3] For the more general definitions that support arbitrary policies, see Appendix A.3.

Corollary 1.1. *If $\phi : \Omega \to Z$ is a Markov state abstraction of MDP $M = (\Omega, A, R, T, \gamma)$ over the policy class Π_ϕ, then the abstract decision process $M_\phi = (Z, A, R^\pi_{\phi,t}, T^\pi_{\phi,t}, \gamma)$ is also Markov.*

We defer all proofs to Appendix A.4.

Theorem 1 describes a pair of conditions under which ϕ is a Markov abstraction. Of course, the conditions themselves do not constitute a training objective—we can only use them to confirm an abstraction is Markov. In Section 3.3, we adapt these conditions into a practical representation learning objective that is differentiable and suitable for learning ϕ using deep neural networks. First, we show why the Inverse Model condition alone is insufficient.

3.2.2 An Inverse Model Counterexample

The example MDP in Figure 2.7 additionally demonstrates why the Inverse Model condition alone is insufficient to produce a Markov abstraction. Observe that any valid transition between two ground states uniquely identifies the selected action. The same is true for abstract states since the only way to reach z_B is via action a_1, and the only way to leave is action a_0. Therefore, the abstraction satisfies the Inverse Model condition for any policy. However, as noted in Section 2.3.1, conditioning on additional history changes the abstract transition probabilities, and thus the Inverse Model condition is not sufficient for an abstraction to be Markov. In fact, we show in Appendix A.4.2 that, given the Inverse Model condition, the Density Ratio condition is actually *necessary* for a Markov abstraction.

3.3 Training a Markov State Abstraction

We now present a set of training objectives for approximately satisfying the conditions of Theorem 1. Since the theorem applies for the policy class Π_ϕ induced by the abstraction, we restrict the policy by defining π as a mapping from $Z \to \Pr(A)$, rather than from $\Omega \to \Pr(A)$. In cases where π is defined implicitly via the value function, we ensure that the latter is defined over abstract states.

3.3.1 Inverse Models

To ensure the ground and abstract inverse models are equal, we consider a batch of N experiences $(\omega_i, a_i, \omega'_i)$, encode ground states with ϕ, and jointly train a model $f(a|\phi(\omega'_i), \phi(\omega_i); \theta_f)$ to predict a distribution over actions, with a_i as the label. This can be achieved by minimizing a cross-entropy loss, for either discrete or continuous action spaces:

$$\mathcal{L}_{Inv} := -\frac{1}{N}\sum_{i=1}^{N} \log f(a = a_i|\phi(\omega'_i), \phi(\omega_i); \theta_f).$$

Note that because the policy class is restricted to Π_ϕ, if the policy is stationary and deterministic, then $I^\pi_{\phi,t}(a|z', z) = \pi_\phi(a|z) = \pi(a|\omega) = I^\pi_t(a|\omega', \omega)$ and the Inverse Model condition is satisfied trivially. Thus we expect $\mathcal{L}_{Inv}$ to be most useful for representation learning when the policy has high entropy or is changing rapidly, such as during early training.

3.3.2 Density Ratios

The second condition, namely that $\frac{P^\pi_{\phi,t}(z'|z)}{P^\pi_{\phi,t}(z')} = \frac{P^\pi_t(\omega'|z)}{P^\pi_t(\omega')}$, means we can distinguish conditional samples from marginal samples equally well for abstract states or ground states. This objective naturally lends itself to a type of contrastive loss. We generate a batch of N sequential ground state pairs (ω_i, ω'_i) as samples of $\Pr(\omega'|\omega)$, and a batch of N non-sequential pairs $(\tilde{\omega}_i, \omega'_i)$ as samples of $\Pr(\omega')$, where the latter pairs can be obtained, for example, by shuffling the ω_i states in the first batch. We assign positive labels ($y_i = 1$) to sequential pairs and negative labels to non-sequential pairs. This setup, following the derivation of Tiao (2017), allows us to write density ratios in terms of class-posterior probabilities: $\delta(\omega') := \frac{\Pr(\omega'|\omega)}{\Pr(\omega')} = \frac{p(y=1|\omega,\omega')}{1-p(y=1|\omega,\omega')}$ and $\delta_\phi(z') := \frac{\Pr(z'|z)}{\Pr(z')} = \frac{q(y=1|z,z')}{1-q(y=1|z,z')}$, where p and q are just names for specific probability distributions.[4] We jointly train an abstraction ϕ and a classifier $g(y|\phi(\omega'), \phi(x); \theta_g)$, minimizing the cross-entropy between predictions and labels y_i:

$$\mathcal{L}_{Ratio} := -\frac{1}{2N} \sum_{i=1}^{2N} \log g(y = y_i|\phi(\omega'_i), \phi(\omega_i); \theta_g).$$

In doing so, we ensure g approaches p and q simultaneously, which drives $\delta_\phi(z') \to \delta(\omega')$.

Note that this is stronger than the original Inverse Model condition, which only required the ratios to be equal in expectation. This stronger objective, when combined with the inverse loss, actually encodes a novel form of the kinematic inseparability conditions from Section 3.1.5, which further helps to avoid representation collapse. (See Appendix A.6 for more details.)

3.3.3 Smoothness

Given a Markov state abstraction ϕ, we can always generate another abstraction ϕ' by adding a procedural reshuffling of the ϕ representation's bits. Since the ϕ' representation contains all the information that was in the original representation, ϕ' is also a Markov state abstraction. However, the new representation may be highly inefficient for learning.

To demonstrate this, we ran an experiment where we optionally relabeled the positions in the 6×6 gridworld domain, and trained two agents: one using the smooth, true (x, y) positions, and one using the non-smooth, relabeled positions. Although both representations are Markov, and contain exactly the same information, we observe that the agent trained on the non-smooth positions performed significantly worse (see Figure 3.3).

[4] For completeness, we reproduce the derivation in Appendix A.5.

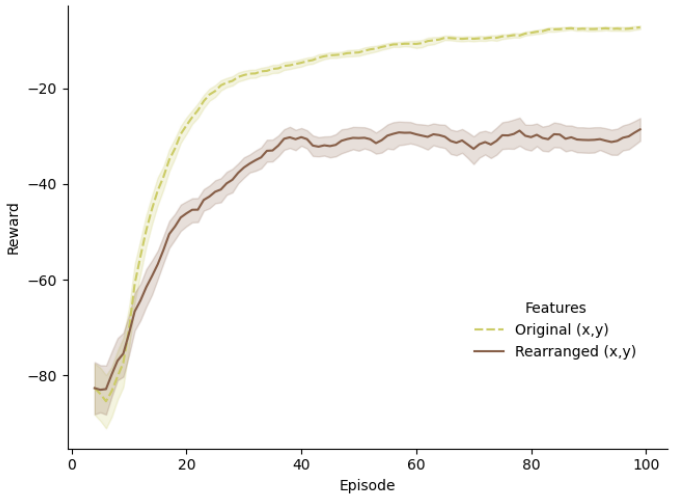


Figure 3.3: Mean episode reward for the 6×6 gridworld navigation task, comparing original (x, y) position with rearranged (x, y) position. (300 seeds; 5-point moving average; shaded regions denote 95% confidence intervals).

To improve robustness and encourage our method to learn smooth representations like those discussed in Section 3.1.6, we optionally add an additional term to our loss function:

$$\mathcal{L}_{Smooth} := (\text{ReLU}(\|\phi(\omega') - \phi(\omega)\|_2 - d_0))^2.$$

This term penalizes consecutive abstract states for being more than some predefined distance d_0 away from each other, thereby encouraging representations to have a high degree of smoothness in addition to being Markov. This approach is similar to the temporal coherence loss proposed by Jonschkowski and Brock (2015).

3.3.4 Markov Abstraction Objective

We generate a batch of experiences using a mixture of abstract policies $\pi_i \in \Pi_C \subseteq \Pi_\phi$ (for example, with a uniform random policy), then train ϕ end-to-end while minimizing a weighted combination of the inverse, ratio, and smoothness losses:

$$\mathcal{L}_{Markov} := \alpha\mathcal{L}_{Inv} + \beta\mathcal{L}_{Ratio} + \eta\mathcal{L}_{Smooth},$$

where α, β, and η are coefficients that compensate for the relative difficulty of minimizing each individual objective for the domain in question.

The Markov objective avoids the problem of representation collapse without requiring reward information or ground state prediction. A trivial abstraction like $\phi(\omega) \mapsto 0$ would not minimize $\mathcal{L}_{Markov}$, because it contains no useful information for predicting actions or distinguishing authentic transitions from manufactured ones.

3.4 Offline Abstraction Learning for Visual Gridworlds

First, we evaluate our approach for learning an abstraction offline for a visual gridworld domain (Fig. 3.1). Each discrete (x, y) position in the 6×6 gridworld is mapped to a noisy image (see Appendix A.7). We emphasize that the agent only sees these images; it does not have access to the ground-truth (x, y) position. The agent gathers a batch of experiences in a version of the gridworld with *no rewards or terminal states*, using a uniform random exploration policy over the four directional actions.

Prior to any reinforcement learning, the agent uses these experiences to train an abstraction function ϕ_{Markov}, by minimizing $\mathcal{L}_{Markov}$ (with $\alpha = \beta = 1$, $\eta = 0$). We visualize the learned 2-D abstract state space in Figure 3.4 (top row) and compare against ablations that train with only $\mathcal{L}_{Inv}$ or $\mathcal{L}_{Ratio}$, as well as against two baselines that we train via pixel prediction and reconstruction, respectively (see Appendix A.9 for additional training runs). We observe that ϕ_{Markov} and ϕ_{Inv} cluster the noisy observations and recover the 6×6 grid structure, whereas the others do not generally have an obvious interpretation. We also observed that ϕ_{Ratio} and $\phi_{Autoenc}$ frequently failed to converge.

Next we froze these abstraction functions and used them to map images to abstract states while running reinforcement learning on the resulting features (specifically using deep Q-networks, DQN; see Mnih et al. (2015)). We measured the learning performance of each pre-trained abstraction, as well as that of end-to-end deep reinforcement learning with no pretraining. We plot learning curves in Figure 3.5. For reference, we also include performance comparisons with a uniform random policy and learning from ground-truth (x, y) position with no abstraction.

Markov state abstractions match the performance of ground-truth position, and beat every other learned representation except ϕ_{Inv}. Note that while ϕ_{Markov} and ϕ_{Inv} perform similarly in this domain, there is no reason to expect L_{Inv} to work on its own for other domains, since it lacks the theoretical motivation of our combined Markov loss. When the combined loss is minimized, the Markov conditions are satisfied. But *even if* the inverse loss goes to zero on its own, the counterexample in Section 3.2.2 demonstrates that this is insufficient to learn a Markov state abstraction.

3.5 Online Abstraction Learning for Continuous Control

Next, we evaluate our approach in an online reinforcement learning setting for a collection of image-based, continuous control tasks from the DeepMind Control Suite (Tassa et al., 2020). Our training objective is agnostic about the underlying RL algorithm, so we use as our baseline the state-of-the-art technique that combines Soft Actor-Critic (SAC) (Haarnoja et al., 2018) with random data augmentation (RAD) (Laskin et al., 2020a). We initialize a replay buffer with experiences from a uniform random policy, as is typical, but before training with RL, we use those same experiences *with reward information removed* to pretrain a Markov

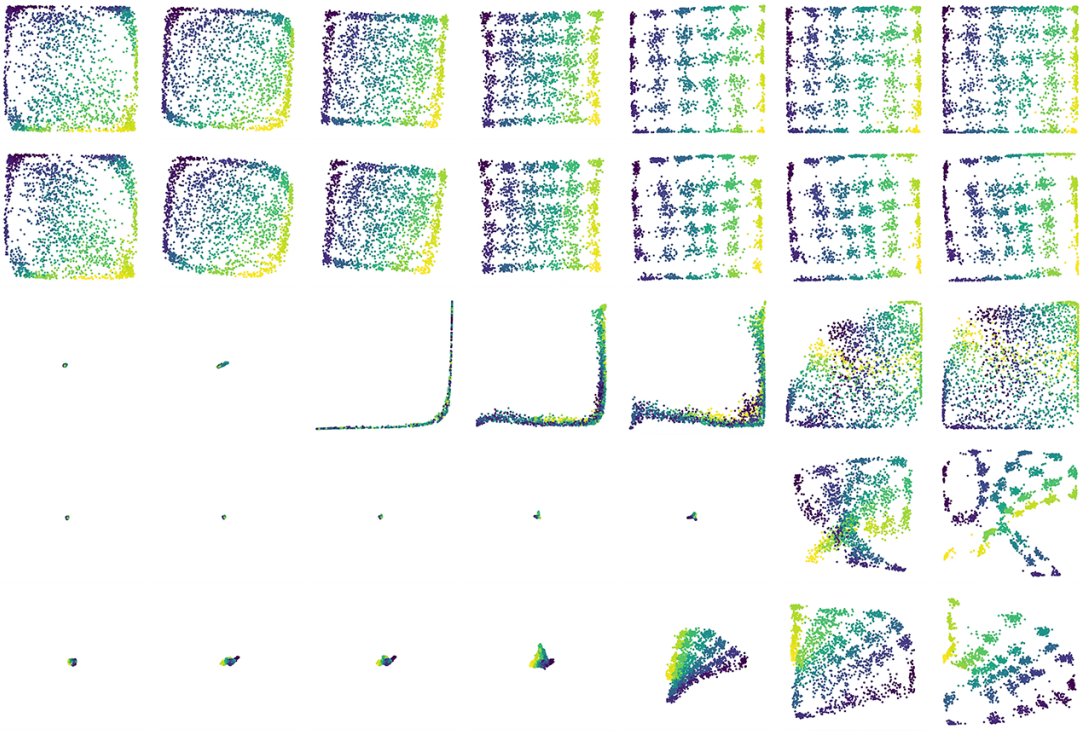


Figure 3.4: Visualization of learning progress for 2-D state abstractions in the 6×6 visual gridworld domain. Each row displays selected times (after 1, 100, 200, 700, 3K, 10K, and 30K steps, progressing from left to right) of a different abstraction learning method (top to bottom): $\mathcal{L}_{Markov}$; $\mathcal{L}_{Inv}$ only; $\mathcal{L}_{Ratio}$ only; autoencoder; pixel prediction. Each point encodes a single observation. Color denotes ground-truth (x, y) position, which is not shown to the agent.

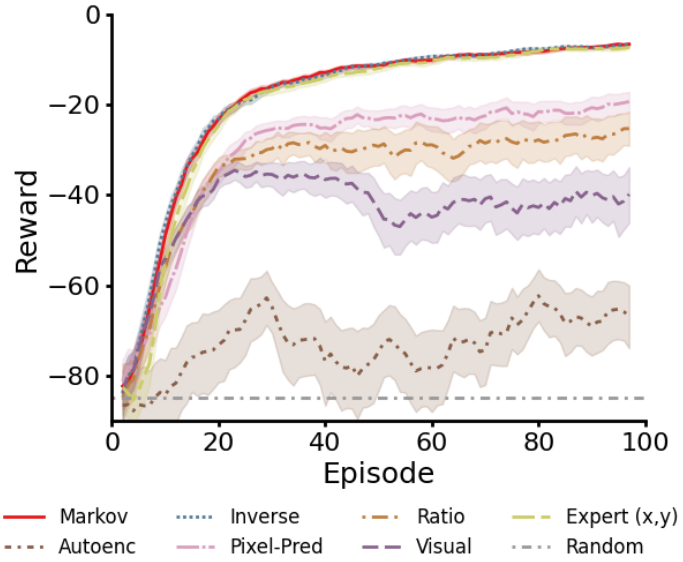


Figure 3.5: Mean episode reward for the visual gridworld navigation task. Markov abstractions significantly outperform end-to-end training with visual inputs, and match the performance of the expert (x, y) position features. (300 seeds; 5-point moving average; shaded regions denote 95% confidence intervals.)

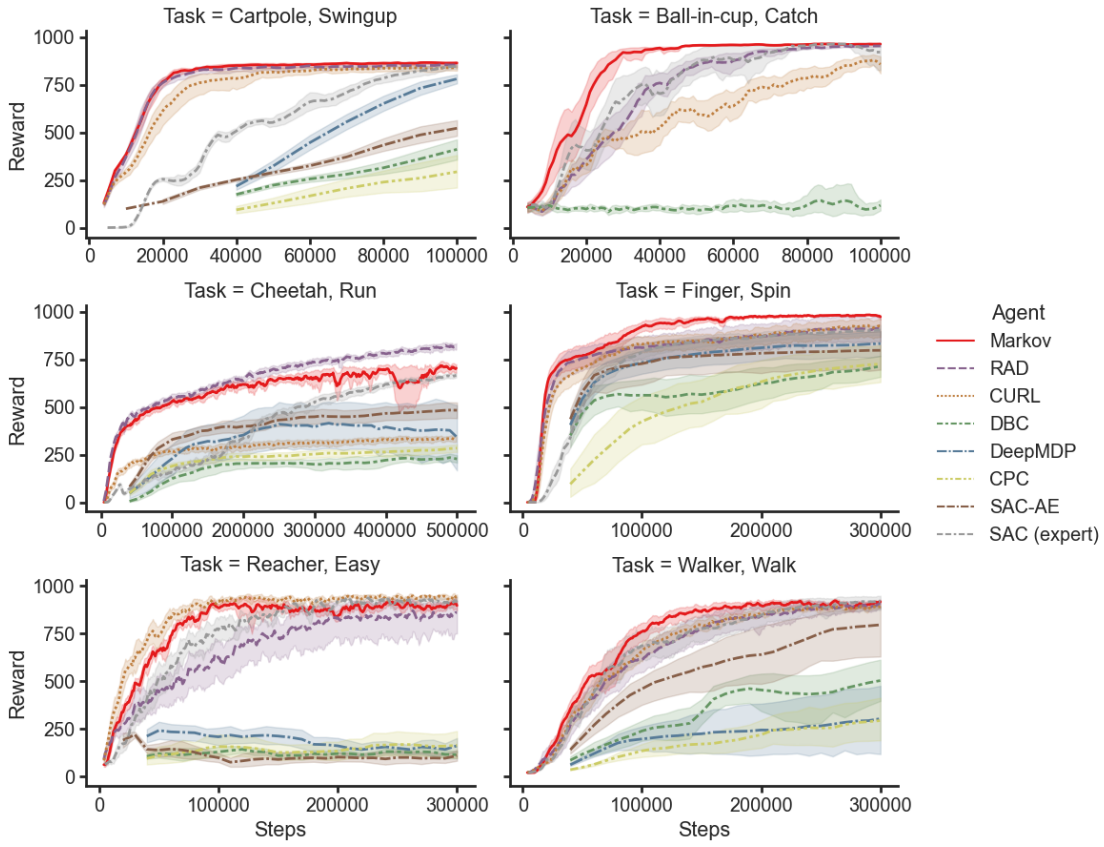


Figure 3.6: Mean episode reward vs. environment steps for DeepMind Control. Adding our Markov objective leads to improved learning performance. (10 seeds; 5-point moving average; shaded regions denote 90% confidence intervals; learning curve data is available at the linked code repository.)

abstraction. We then continue training with the Markov objective alongside traditional RL. (See Appendix A.8 for implementation details).

In Figure 3.6, we compare against an unmodified reinforcement learner (RAD), as well as contrastive methods CURL (Laskin et al., 2020b) and CPC (van den Oord et al., 2018), bisimulation methods DeepMDP (Gelada et al., 2019) and DBC (Zhang et al., 2021), and pixel-reconstruction method SAC-AE (Yarats et al., 2019). As a reference, we also include non-visual SAC with expert features. All methods use the same number of environment steps (the experiences used for pretraining are not additional experiences).

Relative to the unmodified agent (RAD), our method learns faster on four domains and slower on one, typically achieving the same final performance (better in one, worse in one). It performs even more favorably relative to the other baselines, of which CURL is most similar to our method, since it combines contrastive learning with data augmentation similar to that of RAD.[5] Our approach even represents a marginal improvement over a hypothetical "best of" oracle that always chooses the best performing baseline. These experiments show

[5]We ran another experiment with no data augmentation, using a different state-of-the-art continuous control algorithm, RBF-DQN (Asadi et al., 2021), and found similar results there as well (see Appendix A.11 for details).

that when a uniform random policy is insufficient to cover the full state space, the agent can continue to refine its abstract representation using online reward information. Here we see further evidence that explicitly encouraging Markov state abstractions improves learning performance over state-of-the-art image-based RL.

3.6 Conclusion

In this chapter, we described a principled approach to learning abstract state representations that provably results in Markov abstract states, and which does not require estimating transition dynamics nor ground-state prediction. We defined what it means for a state abstraction to be Markov while ensuring that the abstract MDP accurately reflects the dynamics of the ground MDP, and introduced sufficient conditions for achieving such an abstraction. We adapted these conditions into a practical training objective that combines inverse model estimation and contrastive learning to simultaneously explain state transitions and distinguish real transitions from fake ones. Our approach learns abstract state representations, with or without reward information, that capture the Markov structure of the underlying domain and substantially improve learning performance over existing approaches. This chapter relied heavily on the block MDP assumption to guarantee that observations were themselves Markov states. In the next chapter, we will remove that assumption and consider general decision processes with non-Markov observations.

Chapter 4

History Abstractions

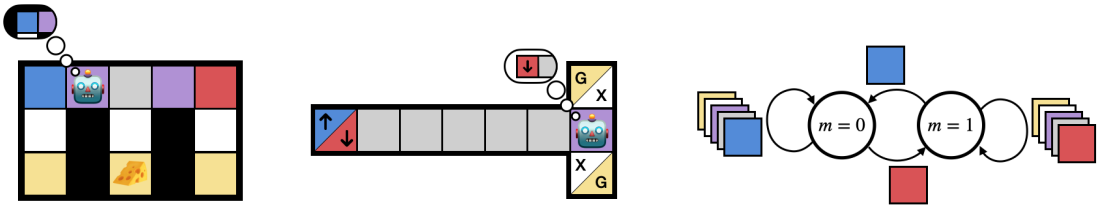
> If you don't know history, it is as if you were born yesterday.
>
> —Howard Zinn

Recall that the Markov assumption has several desirable implications. First, the transition and reward functions have fixed-sized inputs and are therefore easy to parameterize, learn, and reuse. Second, it follows that the policy π need only reactively map from states to actions: $\pi : \mathcal{S} \to \mathcal{A}$, again resulting in a function with a fixed-sized input; similarly for the corresponding value functions $V_\pi(s)$ and $Q_\pi(s, a)$, which are the most common targets for learning. Finally, if the Markov property holds then so does the Bellman equation (2.1), the core equation underlying temporal difference methods, and the foundation upon which the vast majority of reinforcement learning solution methods rest (Sutton and Barto, 2018).

Let us now turn our attention to the more realistic problem of reinforcement learning in non-Markov decision processes (henceforth simply *decision processes*) like the ones in Figure 4.1a. Without some way of accessing the latent state $\mathcal{S}$ governing the behavior of the system, all the relevant functions—transition dynamics T, rewards R, policy π, and value functions V and Q—depend on the agent's entire interaction history h_t. Additionally, the Bellman equation can no longer be expected to hold, leaving the technical appropriateness of most reinforcement learning algorithms uncertain.

History abstraction (see Fig. 4.1b) offers an alternative to learning with unbounded and variable-length histories, but how should we learn a good memory function? As with observation abstractions, the dominant strategy is to allow memory functions to simply emerge over time via end-to-end reinforcement learning with neural networks (Kapturowski et al., 2018). While this can often be effective, there is usually no guarantee that the memory function will be in any way optimal.

In this chapter, we explore a new method for learning memory functions. We introduce the λ-*discrepancy*, the



(a) (b)

Figure 4.1: (a) Cheese Maze (left) and Bakker's T-Maze (right), examples of non-Markov decision processes. The agent must navigate through the environment but can only distinguish squares by their color. The agent will more easily reach the goal (cheese; G) and receive positive reward if it can remember at purple squares whether it previously saw a blue or a red observation. G, X, and cheese are terminal states. (b) A memory function, expressed as a finite state machine, that tests whether a blue or a red observation was observed more recently.

difference between the return predicted by temporal difference learning and a Monte Carlo value estimate, or more generally between TD(λ) with two different values for λ. These quantities are equal for Markov decision processes, but, crucially, different in general decision process. We prove that the λ-discrepancy can be used to reliably identify non-Markov reward or transition dynamics, and propose to use it as a loss function in *memory optimization*, a process where memory functions are optimized to minimize the λ-discrepancy. We empirically demonstrate that using the λ-discrepancy can effectively achieve high performance in general decision processes.

4.1 Learning Memory via the λ-Discrepancy

A memory function allows the agent to define a memory-augmented policy $\pi_\mu : (\Omega \times M) \to A$, which naturally leads to memory-augmented value functions $V_{\pi,\mu} : (\Omega \times M) \to \mathbb{R}$ and $Q_{\pi,\mu} : (\Omega \times M) \times A \to \mathbb{R}$. These augmented value functions ideally should reflect the expected return under policy π_μ. Unfortunately, the memory-augmented analogs of the Bellman equation (2.1) do not converge to the expected return unless $\Omega \times M$ forms a Markov state; in general, the fixed point is biased. We aim to learn a memory function to minimize this bias, which we write in terms of mean-squared action-value error Δ_Q:

$$\Delta_Q := \mathbb{E}_{h\sim\pi_\mu}\left[\sum_{\omega\in\Omega} O(\omega|h)\big(Q_\pi(\omega,a) - \hat{Q}_{\pi,\mu}(\omega,a,h)\big)^2\right], \tag{4.1}$$

where $Q_\pi(\omega, a)$ is the expected return taking action a after observation ω under policy π, and $\hat{Q}_{\pi,\mu}(\omega, a, h)$ estimates that return by first computing a memory state $m = \mu(h)$ and then estimating an action-value function $\hat{Q}_{\pi,\mu}(\omega, m)$. We also consider the corresponding value error $\Delta_V = \mathbb{E}_\pi[\Delta_Q]$.

In Section 2.1.1 we introduced two popular approaches for estimating value functions: Monte Carlo (MC) methods and temporal difference (TD) methods. While these methods have the same fixed point in Markov decision processes, the same does not hold for general decision processes. Moreover, the difference between their estimates is strikingly reminiscent of the action-value error of Eq. (4.1), as we will see below.

MC forms an estimate by directly averaging samples of the return:

$$\hat{Q}_\pi^{MC}(\omega, a) := \mathbb{E}_\pi[G_t|\omega_t = \omega, a_t = a] \approx \frac{1}{N}\sum_{i=1}^{N} G_t^{(i)}[\omega_t = \omega, a_t = a],$$

where the expectation is approximated by sampling entire trajectories under the policy. This is an unbiased estimator of $Q_\pi(\omega, a)$. Meanwhile, temporal difference (TD) methods estimate a value function recursively by bootstrapping off of an existing estimate:

$$\hat{Q}_\pi^{TD}(\omega, a) := \mathbb{E}_{\pi, T_\phi}[R_t + \gamma \hat{Q}_\pi^{TD}(\omega', a')],$$

and this is computed in practice by repeatedly setting $\hat{Q}_\pi^{TD}(\omega, a) \leftarrow R_t + \gamma \hat{Q}_\pi^{TD}(\omega', a')$.

The TD estimator implicitly includes the Markov assumption: the substitution of $\hat{Q}_\pi^{TD}(\omega', a')$ in place of G_{t+1} assumes that conditioning on (ω', a') is sufficient for characterizing the distribution of future returns. TD learns a biased estimate of value because it averages over all trajectories consistent with the observation-action pair (ω', a'), regardless of whether they are compatible with the preceding experiences. By contrast, the MC estimator computes expected return using the remainder of the actual, realized trajectory, and is unbiased.[1] If we take the interpretation that TD is our estimator, and MC is a sample of the return, then the squared difference between TD and MC provides us with an unbiased estimate of the mean-squared action-value error of Eq. (4.1).

While MC is unbiased, it is rarely used in practice because of its high variance. Meanwhile, TD has lower variance, but converges to the wrong fixed point. Hybrid approaches like the TD(λ) estimator smoothly interpolate between TD ($\lambda = 0$) and MC ($\lambda = 1$) and offer a nice middle ground; however, all estimators in this family except $\lambda = 1$ implicitly include the Markov assumption, with the severity of its impact controlled by λ.

Instead of tuning λ to find an acceptable balance of bias and variance, we instead propose a learning signal for any memory function that removes the bias of TD entirely. This learning signal compares estimates generated by TD(λ) for two different values of λ. The mismatch between these estimates is a reliable indicator of when an augmented observation space $\Omega \times M$ is non-Markov.

Our analysis models the decision process as a POMDP. Given a reactive policy π, TD(λ) in that setting converges to:

$$Q_\pi^\lambda = W\Big(I - \gamma T\big(\lambda \Pi^S + (1-\lambda)\Phi W^\Pi\big)\Big)^{-1} R^{SA}, \tag{4.2}$$

where Q_π^λ is an $\Omega \times A$ matrix, and the right hand side is comprised of the following multi-dimensional arrays, or tensors: state weights W ($\Omega \times S$) containing probabilities $\Pr(s|\omega)$; identity I (an $S \times A \times S \times A$ tensor with $I_{sas'a'} = \delta_{ss'}\delta_{aa'}$); latent transition dynamics T ($S \times A \times S$); effective policy over latent states Π^S ($S \times S \times A$); observation function[2] Φ ($S \times \Omega$) containing probabilities $\Pr(\omega|s)$; state-action weights W^Π

[1] We visualize this distinction for T-Maze in Figure B.1 in Appendix B.6.2.

[2] Here we use Φ in place of O, both to avoid potential confusion with 0, and to highlight the possibility that partial observability can also stem from an overly aggressive state abstraction.

($\Omega \times S \times A$) containing probabilities $\Pr(s, a|\omega)$; and latent rewards R^{SA} ($S \times A$). Each product between tensors contracts the adjacent indices, e.g. for tensors A and B, $(AB)_{ijlm} = \sum_k A_{ijk} B_{klm}$ with the exception of the product involving R^{SA}, which contracts two indices. This derivation is given in Appendix B.1, and follows the Markov version by Sutton (1988).

4.1.1 The λ-Discrepancy

It is well known that when the POMDP is fully observable (ie. it is an MDP), all TD(λ) estimators converge to the same Q-function. However, in the partially observable case, we expect a discrepancy between Q-values for two different λ parameters due to the implicit Markov assumption in TD(λ). We call this difference the λ-*discrepancy*.

Definition 3. *The* λ-discrepancy $\Delta Q_\pi^{\lambda_1,\lambda_2}$ *is the difference between two action-value functions estimated by TD(λ) using different values of* λ*:*

$$\Delta Q_\pi^{\lambda_1,\lambda_2} := ||Q_\pi^{\lambda_1} - Q_\pi^{\lambda_2}||.$$

In this work, we investigate this as a measure of non-Markovness and learning signal for resolving partial observability. A useful property of this objective function is that λ-discrepancy is 0 in the fully observable setting. Given that this difference is only present in partially observable environments, it could also be a useful measure and objective for reducing partial observability. For POMDPs, we would expect this measure to be non-zero. This brings us to the following theorem:

Theorem 2. *For any POMDP and for any fixed* λ *and* λ'*, either* $\Delta Q_\pi^{\lambda,\lambda'} \neq 0$ *for almost all policies* π *or* $\Delta Q_\pi^{\lambda,\lambda'} = 0$ *for all policies* π*.*

Proof sketch: We formulate the λ-discrepancy as a function from row-stochastic matrices, the set of matrices whose rows sum to 1, to the reals to show that the λ-discrepancy either vanishes at all policies or is nonzero for almost all policies. The full proof is shown in Appendix B.2.

This theorem suggests that if the λ-discrepancy is non-zero for some policy π, then the λ-discrepancy is a measure of partial observability and could be used as a learning signal to resolve partial observability. While the *almost all* condition may seem problematic, these policies are measure zero in the space of all policies, and can be avoided by small perturbations (for instance, with an ϵ-greedy policy).

We now consider when the λ-discrepancy is 0 for all policies, and show that these POMDPs either have Markov observations or come from a very narrow set of uninteresting POMDPs.

4.1.2 When is the λ-discrepancy zero?

To analyze the case in Theorem 2 where the λ-discrepancy is 0 for all policies, we consider the analytical version of Definition 3:

$$\Delta Q_\pi^{\lambda_1,\lambda_2} = W\Big(A_\pi^{\lambda_1} - A_\pi^{\lambda_2}\Big)R^{SA}, \quad \text{where } A_\pi^\lambda = \Big(I - \gamma T\left(\lambda\Pi^S + (1-\lambda)\Phi W^\Pi\right)\Big)^{-1}. \tag{4.3}$$

The only ways for this to be zero are either when the difference term $A_\pi^{\lambda_1} - A_\pi^{\lambda_2}$ is zero (which we will show implies Markov observations), or in the unlikely event that this difference term is projected away by the outer terms W and/or R^{SA}. We first consider when the two inner terms—which are the only terms that depend on λ—are equal, i.e.:

$$A_\pi^{\lambda_1} = A_\pi^{\lambda_2}. \tag{4.4}$$

In this case the system is a block MDP, where each observation can be produced by only one unique state. This results in a λ-discrepancy of 0:

Lemma 4.1.1. *For any POMDP and any λ, λ', Equation 4.4 holds if and only if the system is a block MDP.*

Proof: See Appendix B.3.

Finally, we consider POMDPs where the difference between A_π^λ is projected away by the outer terms W and R^{SA}. To see how a λ-discrepancy of 0 is unlikely in this case, we first expand Equation 4.2 as a power series

$$\begin{aligned} Q_\pi^\lambda = {} & WR^{SA} + \gamma WT\left(\lambda\Pi^S + (1-\lambda)\Phi W^\Pi\right)R^{SA} + \\ & \gamma^2 WT\left(\lambda\Pi^S + (1-\lambda)\Phi W^\Pi\right)T\left(\lambda\Pi^S + (1-\lambda)\Phi W^\Pi\right)R^{SA} + \dots, \end{aligned}$$

and consider when $||Q_\pi^{\lambda_1} - Q_\pi^{\lambda_2}||$ is zero. This would require cancellation of the terms between the two power series. One simple case of this cancellation is when $\lambda = 0$ and $\lambda' = 1$. We can group the power series expansion by γ^n coefficients:

$$\begin{aligned} ||Q_\pi^0 - Q_\pi^1|| = {} & \gamma\left(WT\Pi^S R^{SA} - WT\Phi W^\Pi R^{SA}\right) \\ & + \gamma^2\left(WT\Pi^S T\Pi^S R^{SA} - WT\Phi W^\Pi T\Phi W^\Pi R^{SA}\right) + \dots \end{aligned}$$

Setting each bracketed term to 0, this results in a contrived POMDP where the expected reward at each time step is equal to the expected rewards given Markovian observation rollouts. This leads to the following lemma.

Lemma 4.1.2. *For any POMDP, $\Delta Q_\pi^{0,1} = 0$ if for all initial observations o^0, actions a^0, and horizons T,*

$$\mathbb{E}(r^T|\omega^0, a^0) = \sum_{\omega^1,\dots,\omega^T} \mathbb{E}(r^T|\omega^T)\mathbb{P}(\omega^1|\omega^0, a^0)\prod_{t=1}^{T-1}\mathbb{P}(\omega^{t+1}|\omega^t).$$

Proof: See Appendix B.4.

More generally, other forms of cancellations are also possible, but also unlikely. This would require the difference of two power series to exactly cancel out for two different values of λ. In addition, this condition has to hold *for all policies* π. While it may be possible for a POMDP to exhibit these conditions, this narrow set of POMDPs are exceedingly unlikely—none of the POMDPs introduced in the following experimental sections exhibit this property, and finding an example where these conditions are true has proven to be difficult.

Through Theorem 2 and analyzing redundant or unlikely cases where λ-discrepancy is 0, we conclude that the λ-discrepancy is very likely to be a useful measure for measuring and reducing partial observability. Theorem 2 also implies that if lambda discrepancy is non-zero, minimizing it with respect to some memory function will result in memory-augmented observations that are Markov. It is also a promisingly practical training objective, since it relies on the very value functions that form the foundation of most RL methods and for which many well-developed learning algorithms exist. In the next section, we demonstrate the efficacy of the λ-discrepancy in reducing partial observability in well-known problems, and show how to use it as an objective function for learning memory functions.

4.2 Memory Optimization

In this section we show that minimizing the λ-discrepancy with respect to a memory function resolves partial observability, allowing for optimal policies with higher returns, in both the learning and the planning settings. We begin with describing the analytical version of the memory optimization algorithm for the planning setting in which the agent has access to the ground-truth MDP dynamics (latent transition function T, latent reward R^{SA}, observation function Φ). The agent minimizes the λ-discrepancy through analytical gradients and optimizes a policy to maximize returns. Next, we describe a sample-based version of the algorithm in the learning setting in which the agent does not have access to this information and instead must compete with existing RNN-based solutions. We empirically show that memory functions learnt through minimizing λ-discrepancy reduce partial observability in a wide array of small partially observable environments and increase optimal policy performance in both the planning and learning settings.

4.2.1 Analytical Memory Optimization

To show the efficacy of using the λ-discrepancy as a signal for reducing partial observability, we first describe an analytical algorithm which calculates and optimizes, in closed form, the λ-discrepancy with respect to a parameterized memory function and policy. We do this by augmenting the original POMDP with a *memory Cartesian product* of the POMDP and our memory function. Full details of this theoretical MDP augmentation are elaborated in Appendix B.6.1, and the practical implementation is in B.6.3. We label action-value functions over observations and memory as $Q^{\lambda}_{\pi_\mu}$.

With our memory augmented value functions, together with Definition 3, we can define a concrete λ-discrepancy measure we are seeking to minimize:

$$\Delta Q^{0,1}_{\pi_\mu} := ||Q^0_{\pi_\mu} - Q^1_{\pi_\mu}||_{\pi_\mu,2} \tag{4.5}$$

where $||\cdot||_{\pi_\mu,2}$ is the L_2 norm weighted by policy π_μ. We note that other norms (such as L_1-norm) or other weightings for each observation could be used as a measure of λ-discrepancy. We discuss our choice of observation weighting in Appendix B.5.

One issue with this objective is that memory augments both value functions for the two different λs. This amounts to trying to optimize two value functions to be closer together - an objective function with two constantly moving targets. Instead, in this work we consider an optimization objective more akin to loss functions with a fixed target. We consider the mean-squared action-value error over observations between a target, the Monte-Carlo returns over observations Q^1_π, and the TD(0) estimator which we call $\hat{Q}^0_\pi$, and take the gradient with respect to this objective:

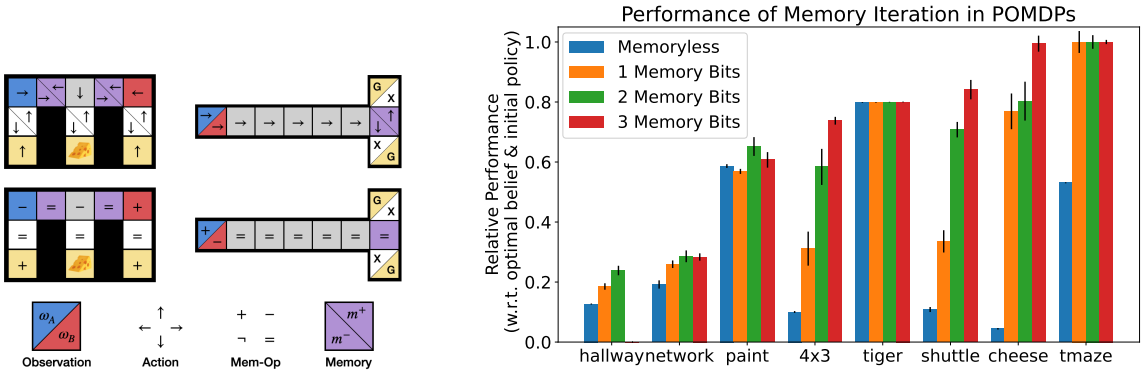
$$\nabla_{\theta_\mu} \Delta Q^{0,1}_\pi = \nabla_{\theta_\mu} ||\hat{Q}^0_\pi - Q^1_\pi||_{\pi,2}, \tag{4.6}$$

where $\hat{Q}^0_\pi = \sum_{m \in M} p(m|\cdot)\hat{Q}^0_{\pi_\mu}$ is the weighted sum of memory-augmented TD(0) action-value functions, and $p(m|\cdot)$ is the probability of seeing some memory state conditioned over each observation. This formulation of the objective function looks to explicitly minimize the TD bias introduced in Section 4.1 and Equation (4.1). It keeps the Monte-Carlo action-value function fixed and independent of memory, and only updates the memory-dependent TD(0) action-value function.

In the following sections, we present analytical results on partially observable environments, where the gradient of this objective function is calculated analytically with respect to the ground-truth dynamics of the environment. To calculate this gradient analytically, we can use any auto-differentiation package. In this work, we use the JAX package (Bradbury et al., 2018) to efficiently do so.

4.2.1.1 Learning Memory Functions

As a case study, we consider the classic partially observable environment *Bakker's T-Maze*, shown in Figure 4.1a, as an example of memory learning. Figure 4.2a shows one of the potential memory functions learnt with the above gradient calculations. This memory function was learnt with a fixed policy, as shown in the same figure: the agent always traverses to the right until it reaches the end of the hallway, and at the T-junction picks the `up` action with probability $\frac{2}{3}$, or `down` otherwise. To learn the memory function, we repeatedly calculate gradients and use them as updates to our memory function until λ-discrepancy is minimized. In this setting, the learnt memory function allows the agent to represent its ground-truth state optimal policy: the function toggles the memory state according to the agent's initial observation of whether the goal is in the north or south end of the T-junction. For the corridor observations, μ is the identity function, which allows the agent to remember the initial observation at the T-junction. Appendix B.6.2 elucidates further environment and experimental details. We call the above process of descending the λ-discrepancy gradient



(a) (b)

Figure 4.2: (a) Learnt policies (top) and memory functions (bottom) for Cheese Maze (left) and T-maze (right), using analytical memory optimization. Split squares denote observation- or memory-dependent behavior. Memory operations $+$, $-$, $\neg$ (unused), and $=$ respectively set, reset, negate, and retain the memory bit. (b) Results for analytical memory optimization. Performance is normalized between a belief state policy (learned with a POMDP solver (Cassandra, 2003)) and a random policy performance. Performance is calculated as the analytical average value weighted by the start state distribution. Error bars are standard error of the mean over 30 seeds.

to convergence *memory improvement*. With this memory-learning mechanism in place, we can utilize these learnt memory functions to augment the agent's observations. In the following sections, we present analytical results demonstrating the efficacy of these learnt memory function to resolve partial observability and improve the agent's policy. We compare policy improvement over the original partially observable environment and one in which observations are augmented using the learnt memory function.

4.2.1.2 Analytical Memory Optimization Planning Experiments

We present results for a number of classic partially observable environments, including Bakker's T-maze, the Tiger Problem Cassandra et al., 1994, Paint (Kushmerick et al., 1995), Cheese Maze, Network, Shuttle (Chrisman, 1992), the 4×3 maze (Russell and Norvig, 1995) and the Hallway problem (Kaelbling et al., 1998). Results shown in Figure 4.2b show the normalized performance of both the converged memoryless policy (blue) learnt through LSPI and the converged memory-augmented policies for memory functions with varying amounts of memory bits learnt through memory optimization. Performance is normalized between the belief-state policy learnt from a POMDP solver (Cassandra, 2003) and the performance of a random policy.

Even with only a single bit of memory, we see large performance increases for all but two of the POMDPs we test on. These results suggest that reducing the λ-discrepancy is a useful objective for learning memory functions to resolve partial observability in many partially observable domains. Note that the Hallway environment

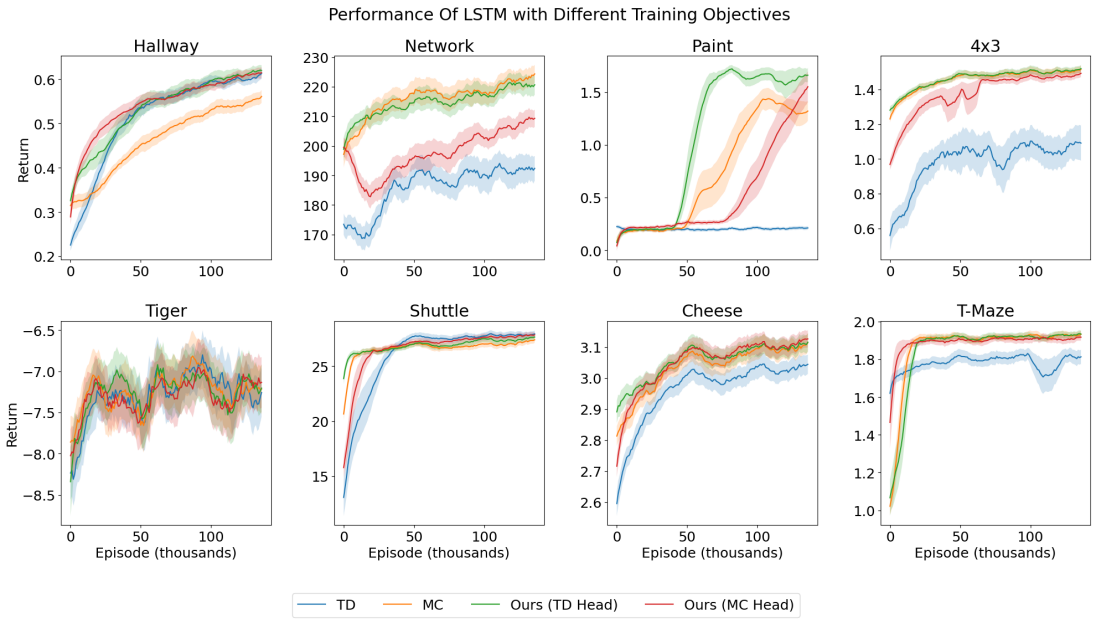


Figure 4.3: Learning curves for LSTM neural network trained with various loss functions. Our model includes both TD and MC value heads and uses an ϵ-greedy policy w.r.t. one of the heads.

4.2.2 Sample-Based Memory Optimization Learning Experiments

We can approximate the lambda discrepancy minimization objective with recurrent neural networks, simply by training two value function heads at the same time. We decompose the optimization objective $\mathcal{L}$ as:

$$\mathcal{L} = (v_1 - \hat{v}_1)^2 + (v_0 - \hat{v}_0)^2 = (v_1 - v_0)^2 + 2(v_1 - \hat{v}_0)(v_0 - \hat{v}_1) + (\hat{v}_1 - \hat{v}_0)^2,$$

where $\hat{v}_0$ and $\hat{v}_1$ are the two value function heads, estimating the TD and MC value functions v_0 and v_1, respectively. The final term of this decomposition is the lambda discrepancy term we are trying to minimize. However, the construction of the neural network does not allow for us to independently tune the importance of the lambda discrepancy, so we optionally include a separate lambda discrepancy loss term, which we compute as the squared difference between the two heads. This allows us to tune the coefficient of the discrepancy term as necessary.

We present results on using the λ-discrepancy to learn implicit memory functions with neural networks in Figure 4.3. Our general network architecture consists of a single LSTM layer with linear output layers for each head. The input to the LSTM layer is a one-hot vector of observations for all environments. For experiments labelled as TD or MC, we have a single head that approximates either TD or MC returns, while for experiments labelled as "ours", we have two heads, each with a separate linear layer, and we tune the coefficient of the λ-discrepancy across all domains. We consider two such λ-discrepancy models, corresponding to which of the two heads' value estimates we use to select actions. We find that using both the MC and TD heads significantly improves results over using either the TD or MC heads. See Appendix B.7 for results on RockSample (Smith and Simmons, 2004), a more complex domain where our method again outperforms both baselines.

4.3 Conclusion

In this chapter, we explored a general framework for learning history abstractions in POMDPs. We introduced a new quantity, the λ-discrepancy, and proved several theoretical properties that establish it is as a reasonable optimization objective for learning effective memory functions. We evaluated an analytical approach to memory optimization based on minimizing the λ-discrepancy for several classical POMDP problems, and showed that this process leads to effective memory functions that support learning better policies. We also provided a practical training procedure for learning memory functions online with recurrent neural networks, and showed that the λ discrepancy improved performance there as well.

In the next two chapters, we will shift the target of our abstractions from the agent's inputs (observations) to its outputs (actions).

Chapter 5

Action Abstractions

> If you are curious, you'll find the puzzles around you. If you are determined, you will solve them.
>
> —Ernö Rubik

Heuristic search is only useful with a good heuristic. In classical planning, much work has gone into the development of domain-independent methods that automatically construct heuristics to exploit as much problem structure as possible from the formal PDDL problem description (Bonet and Geffner, 2001; Hoffmann and Nebel, 2001; Helmert, 2006; Helmert and Domshlak, 2009; Helmert et al., 2014; Pommerening et al., 2015; Keyder et al., 2014; Domshlak et al., 2015). However, as we've already discussed, black-box planning offers no formal domain description to exploit, and planners are therefore limited to less-informed heuristics like goal counting, which is often no better than exhaustive search.

We begin by examining why goal counting becomes uninformative for certain sets of actions. We show that both goal-count accuracy and planning efficiency are linked to how many state variables actions can modify at once. Our investigation suggests a compelling strategy for improving the usefulness of the goal-count heuristic: learning *focused* macro-actions that modify as few variables as possible, so as to align with the assumptions made by the goal-count heuristic. This approach also seems well-aligned with human problem solving, for example, among expert Rubik's cube solvers, where focused macros are essential for the most efficient planning strategies.

We describe a method for discovering focused macro-actions and test it on several classical planning benchmarks, restricting our attention to satisficing solutions—that is, finding feasible plans, rather than optimal ones—with the goal of minimizing the number of world-model queries. Our learned macro-actions enable reliable and efficient planning, making dramatically fewer model queries and improving solve rate on most domains. Our approach is designed to improve the goal-count heuristic, but it is compatible with more

sophisticated black-box planning techniques as well—with similar improvement. In some cases, black-box planning with focused macros is even competitive with approaches that have access to much more detailed problem information.

5.1 Goal-Count Accuracy and Effect Size

Recall that the goal-count heuristic, which we will denote $\#g$, is defined in terms of the problem-specific goal, G. For any goal condition G, $\#g(s)$ counts the number of variables in state s whose values differ from those specified in G, with $\#g(s) = 0$ if and only if s satisfies G. A well-known downside of the goal-count heuristic is its dependence on the size of G. In the extreme case where $|G| = 1$, the goal-count heuristic only separates goal states from non-goal ones. Nevertheless, due to the relative lack of information in black-box planning, the goal-count is often the only goal-aware heuristic available. Other planners may add additional components, such as state novelty (Francès et al., 2017), but the black-box versions of those planners still rely on goal counting at their core.

The goal-count heuristic implicitly treats each state variable as an independent subgoal. There are two ways to satisfy this assumption exactly. The first is if each subgoal can be achieved in one step without modifying any other state variable. The second, more general way, explored by Korf (1985b), is if each subgoal can be achieved in one step (possibly modifying other state variables) and the subgoals are serializable—i.e. there is an ordering of the state variables that retains previously-solved subgoals when solving new ones.

In general, an action can of course change many state variables, and the problem representation may not allow the subgoals to be serialized—both of which can cause the goal-count heuristic to be uninformative. However, for a heuristic to be useful, it does not need to be perfect; it simply needs to be *rank correlated* with the distance to the goal: higher true distances should correspond to higher heuristic values (Wilt and Ruml, 2015). When the heuristic is perfectly rank correlated, there is a monotonic relationship between heuristic and true cost, and best-first search will always expand nodes in order of their true distance from the goal.

We hypothesize that if each action modifies only a small number of state variables, the problem will better match the assumptions of the goal-count heuristic, and thus the heuristic and true goal distance will be more positively rank correlated. We informally say such actions have “focused" effects, and we formalize this idea with the following definitions:

Definition 4. The *effect size of an action* is the maximum number of state variables whose values change by executing the action, over all states where the action is applicable.

Definition 5. The *effect size of a macro-action* is the maximum number of state variables, measured at the end of macro-action execution, that are different from their starting values, over all states where the macro-action is applicable, even if additional variables were modified during execution.

If our hypothesis above is correct, we expect the goal-count heuristic to be more accurate for domains where

actions have smaller effect size, and we further expect this to lead to an improvement in planning efficiency. In the following experiment, we see better rank correlation between heuristic and true distance for domains whose actions have low average effect size, and we see that this leads to an approximately exponential improvement in planning efficiency.

5.1.1 The Suitcase Lock Domain

To study the relationship between effect size and planning efficiency, we introduce the Suitcase Lock domain. The Suitcase Lock is a planning problem whose solution requires entering a combination on a lock with N dials, each with M digits, and $2N$ actions, half of which increment a deterministic subset of the dials (modulo M) and the other half of which decrement the same dials (see Figure 5.1). For each problem instance, a start state, goal state, and (fixed) action set are generated randomly, and a parameter $\overline{k}$ controls the mean effect size across all actions. This allows us to examine action effect size while holding other problem variables constant. For implementation details, see Appendix C.1 of the supplementary materials.

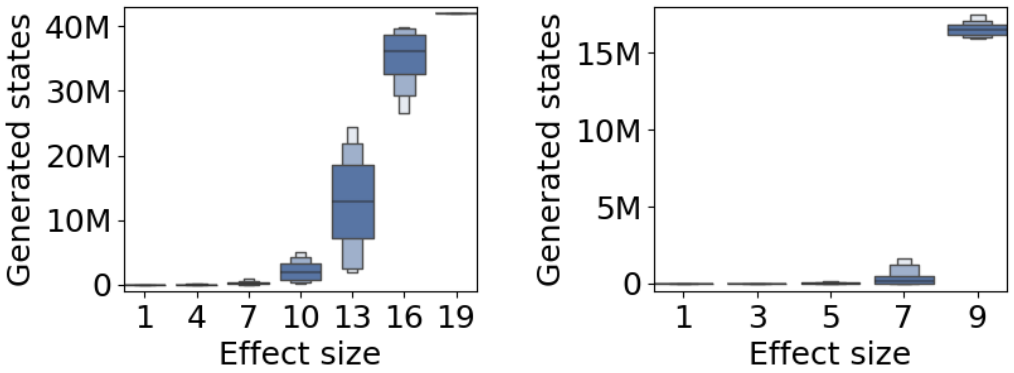
Figure 5.1: Suitcase Lock with $N = 4$.

5.1.1.1 Focused Actions Improve the Goal-Count Heuristic

We first investigate the accuracy of the goal-count heuristic for two small Suitcase Lock problems. For each possible effect size, we compute the true distance between all pairs of states, and compare the results with the goal-count, treating the second state of each pair as the goal. We compute the average heuristic value for each true distance, and then compute the Pearson correlation and Spearman rank correlation coefficients between heuristic and distance. The results are shown in Table 5.1, where we see that actions with more focused effects (i.e. lower $\bar{k}$) lead to significantly higher correlation.

Effect Size	**N=10, M=2**		**N=5, M=4**	
$\bar{k}$	ρ_P	ρ_S	ρ_P	ρ_S
1	1.000	1.000	0.775	0.760
2	0.200	0.179	0.263	0.226
3	0.110	0.092	0.046	0.018
4	0.060	0.041	0.000	-0.044
5	0.020	0.013	–	–
6	0.000	-0.007	–	–
7	0.000	0.001	–	–
8	0.000	-0.001	–	–
9	0.000	0.005	–	–

Table 5.1: Correlation results between the goal-count heuristic and true distance for Suitcase Lock. Actions with smaller effect size ($\bar{k}$) lead to significantly higher Pearson's correlation (ρ_P) and Spearman's rank correlation (ρ_S) coefficients.



$N = 20, M = 2$ $N = 10, M = 4$

Figure 5.2: Generated states vs. effect size for Suitcase Lock.

5.1.1.2 Focused Actions Improve Planning Efficiency

Next, we run two planning experiments using the goal-count heuristic and greedy best-first search (GBFS). To evaluate planning efficiency, we measure the number of generated states needed to solve each instance, since we care about feasibile plans, rather than optimal ones.

Figure 5.2 shows an approximately exponential relationship between effect size and planning time. When $M = 2$ and $k = 1$, the goal-count heuristic is exactly equal to the cost, and GBFS can generate at most N^2 states before finding the goal. By contrast, when $k = (N - 1)$ the heuristic is maximally uninformative, and GBFS may generate $N \cdot 2^N$ states in the worst case, since it cannot expand any nodes with heurstic value k until it has expanded all nodes with heuristic value $< k$. This exponential trend appears to hold even when state variables are not binary.

These results on the Suitcase Lock domain suggest that reducing effect size is a viable strategy for improving planning efficiency. To further investigate this idea, we propose a method for learning macro-actions with low effect size.

5.2 Learning Macros with Focused Effects

We search for macro-actions using best-first search (BFS) with a simulation budget of B_M state transitions. We start the search at a randomly generated state, and the search heuristic is macro-action effect size—or infinity if the macro-action modifies zero variables—plus the number of primitive actions in the macro. Technically, we would need to evaluate each macro from every valid state to determine its effect size, which is clearly infeasible. In practice, we simply measure each macro's effect size once and assume it doesn't change (although we could easily relax this assumption by running the macro from multiple states).

We save the N_M macro-actions with the lowest effect size, and ignore duplicate macro-actions that have the

Algorithm 1 Learn macro-actions with focused effects

Input: Starting state s_0, number of macro-actions N_M, number of repetitions R_M, search budget B_M
Output: List of macro-actions L_M

Define $g(m) := \text{length}(m)$

Define $h(s) := \begin{cases} |\text{net\_effects}(s - s_0)| & \text{if } > 0, \\ \infty & \text{otherwise} \end{cases}$

Define $f(s, m) := g(m) + h(s)$

where m is the macro (i.e. action sequence) from s_0 to s

Let L_M be an empty list of macro-actions
Let Q be a (max) priority queue of size N_M / R_M
for repetition r in $\{1, ..., R_M\}$ **do**
    Run best-first search (BFS) from s_0 with budget B_M / R_M, minimizing heuristic $f(s, m)$
    for each state s_i and macro m_i visited by BFS **do**
        Store m_i in Q, with priority $h(s_i)$
// When Q becomes full, the action sequences
// with largest h-score will get evicted first
    end for
    Add each unique macro in Q to L_M
    Clear Q
    $s_0 \leftarrow$ new random state, such that none of the macros in L_M can run
    if s_0 is None **then**
        break
    end if
end for
return L_M

same net effect. To encourage diversity of macros, we can optionally repeat the search R_M times, each time generating a new random starting state in which none of the existing saved macro-actions are valid, or until we fail to find such a starting state.[1] This ensures that we still find macros that apply in most situations, even if there are constraining preconditions.

The pseudocode for this procedure is in Algorithm 1. We use m_i to denote the macro whose action sequence generated state s_i. Consider an example with two primitive actions a_1 and a_2, where BFS starts at state s_0. Expanding s_0, the action a_1 generates s_1, and a_2 generates s_2. Expanding s_1, a_1 generates s_3, and a_2 generates s_4. Thus macro m_4, corresponding to state s_4, would be the action-sequence $[a_1, a_2]$, and we would evaluate its net effect by comparing s_4 with s_0.

[1] Finding such a state may be as hard as planning, unless the simulator can be reset to generate new starting states; however, in practice, a random walk is often sufficient (see appendix, Note C.2.2.1).

5.3 Experiments

We evaluate our method by learning macro-actions in a variety of black-box planning domains and subsequently using them for planning. We use PDDLGym (Silver and Chitnis, 2020) to automatically construct black-box simulators from classical PDDL planning problems. Additionally, we use two domain-specific simulators (for 15-puzzle and Rubik's cube) that have a different state representation to show the generality of our approach. See the appendix for implementation details and a discussion of how we selected the various macro-learning hyperparameters (sections C.2 and C.5, respectively).

We select the domains to give a representative picture of how the method performs on various types of planning problems. For PDDLGym compatibility reasons, we restrict the domains to those requiring only `strips` and `typing`. For the domain-specific simulators, we select 15-puzzle and Rubik's cube in particular, because they present opposing challenges for our macro-learning approach. In 15-puzzle, primitive actions have very focused effects (each modifies only the blank space and one numbered tile), but naively chosen macro-actions tend to have much larger effect sizes, and both primitive actions and macros have state-dependent preconditions. In Rubik's cube, actions and macros have no preconditions, but primitive actions are highly non-focused (each modifies 20 of the simulator's 48 state variables) and the state space is so large ($\sim 4.3 \times 10^{19}$ unique states (Rokicki, 2014)) that black-box planning is unable to solve the problem efficiently.

5.3.1 Methodology

For each planning domain, we generate 100 problem instances with unique random starting states and a fixed goal condition.[2] All problem instances share the same state space, and the planner has access to the simulator function, the action applicability function, a vector of state information, and the goal condition. We emphasize that although the PDDLGym domains are specified using PDDL, the planner never sees the PDDL during either macro search or planning.

We learn focused macro-actions as described in Sec. 5.2 and add them to the set of primitive actions, which ensures that the same set of states can still be reached. These macros are then used to update the simulator and action applicability function, allowing the learned macros to execute in a single step for improved computational efficiency.[3] Note that updating the simulator in this way does not reduce search effort, only time. Even if the primitive actions in a macro were simulated one-by-one, the intermediate states are neither stored nor explored, and hence do not count towards the number of generated states.

The macros are learned once, for the first problem instance, and then reused on all remaining problem instances for that domain. In general, it can be challenging to incorporate macros into any planning algorithm, since one must weigh their search benefits against the increased branching factor. For simplicity, our experiments fixed

[2] All problem instances are included in the code repository.

[3] See Appendix C.3 for details on how we update the simulator and action applicability function to incorporate the learned macros.

			GBFS(A)		GBFS(A+M)		BFWS(A)		BFWS(A+M)		LAMA(A)	
Domain	N_M	B_M	Gen	Sol	Gen	Sol	Gen	Sol	Gen	Sol	Gen	Sol
Depot	8	50K	58275.9	**0.74**	**55132.4**	0.60	75966.9	**0.48**	**72205.8**	0.34	46620.9	1.00
Doors	8	5K	3050.7	1.00	**512.6**	1.00	4660.9	1.00	**3057.3**	1.00	293.0	1.00
Ferry	8	5K	1875.8	1.00	**1151.4**	1.00	1209.9	1.00	**1163.5**	1.00	699.8	1.00
Gripper	8	5K	7314.8	1.00	**6277.0**	1.00	44945.9	1.00	**6295.9**	1.00	6493.1	1.00
Hanoi	8	100K	78433.6	0.78	**6358.8**	**1.00**	63455.2	1.00	**3365.9**	1.00	65496.4	1.00
Miconic	8	5K	7559.4	1.00	**1907.1**	1.00	10269.2	1.00	**1884.3**	1.00	1316.7	1.00
15-Puz.	192	32K	30840.5	1.00	**4952.4**	1.00	109425.2	1.00	**6290.1**	1.00	–	–
Rubik's	576	1M	>2M	0.00	**171.3K**	**1.00**	>2M	0.00	**163.8K**	**1.00**	9.13M	1.00

Table 5.2: Black-box planning results for PDDLGym-based simulators (top), and domain-specific simulators (bottom). (A) - primitive actions only; (A+M) - primitive actions + focused macros; N_M - number of macros; B_M - macro-learning budget; Gen - generated states; Sol - solve rate; (bold) - best performance of each planner. The efficiency of both GBFS and BFWS(R^*_G) are improved by adding focused macros. Note that LAMA is an informed planner with access to much more information than black-box planners, and is only included for reference.

the number of macros N_M (see Table 5.2), but in principle N_M could be chosen automatically based on which macros reduce the problem's average effect size.

To solve each planning problem, we use greedy best-first search (GBFS) with the goal-count heuristic and compare performance with the additional learned macro-actions versus with primitive actions alone. We measure planning efficiency as the number of simulator queries that the planner makes before finding a plan. This choice of performance metric is the most natural fit for black-box planning, and it allows for fair comparisons of algorithms across different implementation languages and hardware configurations.

In Table 5.2, we show the average solve rate and number of generated states (i.e. simulator queries) for each domain. Since we only pay the macro-learning cost B_M for the first problem instance, we can in principle amortize this cost over the total number of problem instances. (Note that the B_M values reported in the table are non-amortized and are separate from the number of generated states.) Except in the case of Depot, we see that planning with focused macros increases solve rate and improves planning efficiency by up to an order of magnitude versus planning with primitive actions alone. In Rubik's cube, focused macros still perform better, even if we account for the *entire* macro-learning budget.

5.3.2 Comparisons with Other Planners

In addition to greedy best-first search (GBFS) with the goal-count heuristic, we also evaluate our method in conjunction with Best-First Width Search, or BFWS (Lipovetzky and Geffner, 2017), a family of search algorithms that augment their search heuristic with a novelty metric computed using Iterated Width (IW) search (Lipovetzky and Geffner, 2012).

We specifically use the best-performing black-box planning version of BFWS: BFWS(R^*_G) (Francès et al., 2017). This version starts by running IW up to two times, with increasing precision, to generate a set R^*_G of goal-relevant atoms. During search, each state s is evaluated based on how many relevant atoms were satisfied

at some point along the path to s. This forms a relevance count $\#r(s)$, which is combined with the goal-count $\#g(s)$ to compute the novelty width metric $w_{\#r,\#g}$. The algorithm runs GBFS using heuristic $(w, \#g, c)$, evaluating nodes first by width, breaking ties with $\#g$, and then breaking further ties with c, the cost to reach the node.

We ran BFWS on each domain and measured its planning efficiency (see Table 5.2). We followed Lipovetzky and Geffner (2017) and limited the width precision to $w \in \{1, >1\}$ on Depot and Rubik's cube to save computational resources.

Again we find that focused macros substantially improve planning efficiency, likely because the heuristic still uses goal counting at its core. Surprisingly, we found that BFWS did not perform significantly better than the primitive-action GBFS baseline. In fact, comparing against GBFS, we observe that focused macros alone are more beneficial for planning than the more sophisticated novelty-based heuristic.

As a point of reference, we also compared against LAMA (Richter and Westphal, 2010) which has full access to a declarative representation of the problem—information far beyond what is available to black-box planners. We ran the first iteration of LAMA on the same problems we used with PDDLGym, as well as a SAS$^+$ representation of the Rubik's cube, adapted from Büchner (2018). On a different PDDL version of Rubik's cube, LAMA failed to complete the translation step before running out of memory (16GB). We find our method is competitive with LAMA, across the majority of domains, despite the fact that LAMA has access to more information. On the 100 hardest Rubik's cube problems from Büchner (2018), which neither primitive-action baseline can solve, LAMA generates 9.1 million states on average, whereas our approach generates only 171 thousand.

5.3.3 Comparison with Random Macros

One might wonder whether the improvements in planning efficiency are due to the macros' focused effects, or simply the fact that we are using macros at all. To isolate the source of the improvement, we conducted a second experiment using 15-puzzle and Rubik's cube. Here we compared the focused macro-actions against an equal number of "random" macro-actions of the same length, which were generated (for each random seed) by selecting actions uniformly at random from the valid actions at each state.

We present the results in Table 5.3, as well as Figures 5.3a and 5.3b, where we observe that random macros perform significantly worse than both the primitive actions and the learned focused macros. In both domains, random macros also consistently had larger effect sizes than focused macros. Figure 5.3c shows a visualization of Rubik's cube macro effect size versus macro length. We suspect the higher planning cost of random macros is partly due to their increased effect size.

		Generated States	Remaining Errors ($\#g$)	Solve Rate
15-Puz.	Primitives only	30840.5	0.0	1.0
	Random macros	72542.3	0.0	1.0
	Focused macros	**4952.4**	0.0	1.0
Rubik's	Primitives only	>2M	11.8	0.0
	Random macros	>2M	16.4	0.0
	Focused macros	**171331.4**	**0.0**	**1.0**
	Expert macros	30229.1	0.0	1.0

Table 5.3: Planning results for 15-puzzle and Rubik's cube comparing different action spaces. Random macros perform significantly worse than both primitive actions and focused macros. Trials with macros also include the primitive actions.

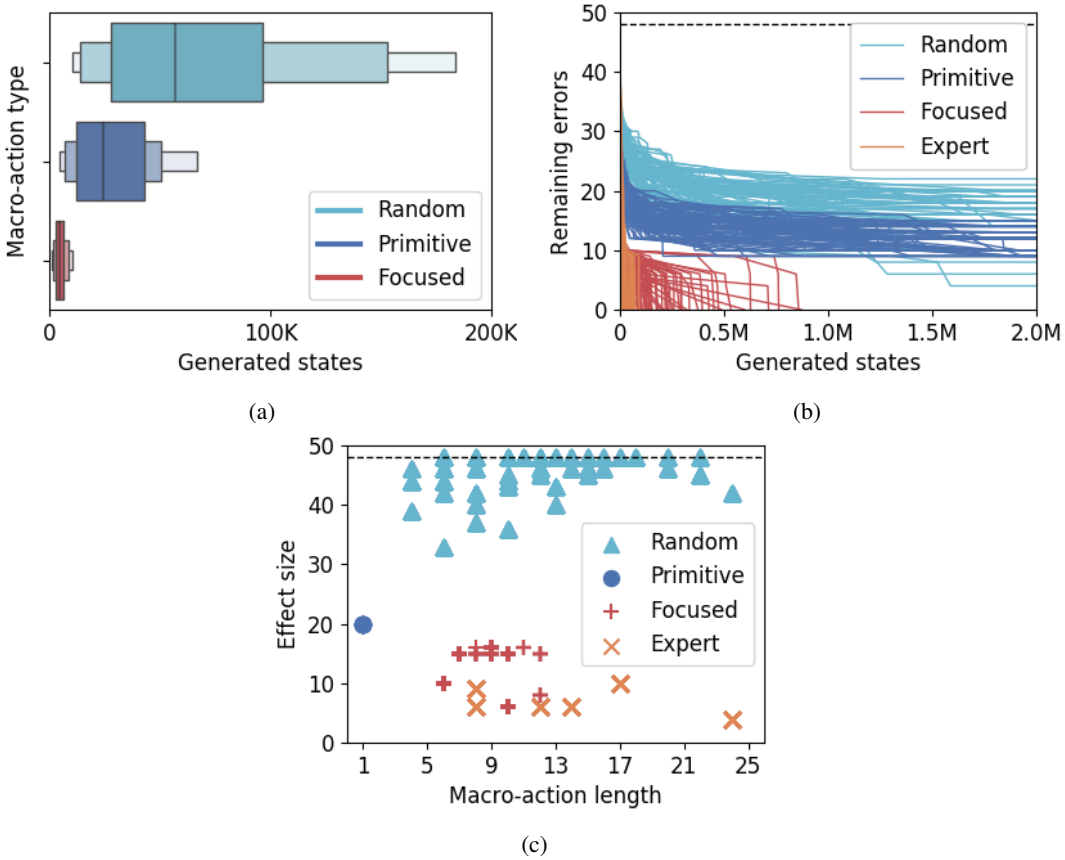


Figure 5.3: (a) 15-puzzle planning efficiency by macro type. Adding focused macros leads to a significant performance improvement over primitive-actions alone. Random macros have the opposite effect. (b) Rubik's cube planning performance by macro type. The vertical axis represents the best observed goal-count value for the number of generated states on the horizontal axis. (c) Effect size vs. length of Rubik's cube macro-actions, by type. (Some points overlap.)

5.3.4 Examining Expert Macros in Rubik's Cube

Expert human "speedcubers" use macro-actions to help them manage the Rubik's cube's highly non-focused actions. In speedcubing, the goal is to solve the cube as quickly as possible, without necessarily finding an optimal plan. Most speedcubers learn a collection of macro-actions (called "algorithms" in Rubik's cube parlance) and then employ a strategy for sequencing those macro-actions to solve the cube. Expert macro-actions tend to affect only a small number of state variables, and proper sequencing enables speedcubers to preserve previously-solved parts of the cube while solving the remainder. Common solution methods typically involve multiple levels of hierarchical subgoals and produce plans approximately twice as long as optimal.

As a benchmark, we consider a simplification of the most common expert strategy, where macros are composed of just primitive actions. We select a set of six hand-coded, expert macro-actions to perform various complementary types of permutations.[4] We visualize one of these macro-actions, which swaps three corner pieces, in Figure 5.4a. Since our simulator uses a fixed cube orientation, we consider all 96 possible variations of each macro (to account for orientation, mirror-flips, and inverses), resulting in 576 total macros—the same number used for the random macro and focused macro trials.

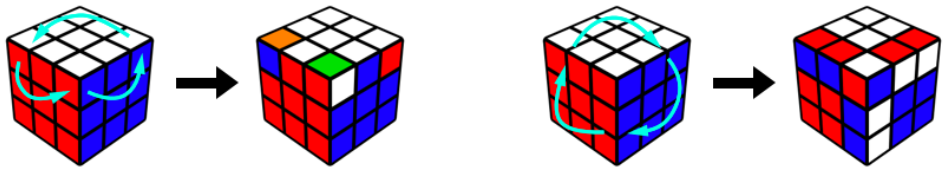
In Figure 5.3c, we plot the effect size and length of each macro, labeled by macro type. We can see that the focused macros have significantly smaller effect size than primitive actions or random macros, and begin to approach the effect size of the expert macro-actions. We also note that the focused macros are somewhat shorter on average than the expert macros, and we suspect that increasing the search budget would result in learning macros with even smaller effects.

In Table 5.3 and Figure 5.3b, we compare planning with the expert macros against the other macro types and see that while planning with the expert macros is the most efficient, the learned, focused macros are not far behind. By contrast, the random macros and primitive actions never solved the problem within the simulation budget. We also found that the average solution length for focused and expert macro-actions was about an order of magnitude longer than typical human speedsolve solutions (378 and 319 primitive actions, respectively, vs. ~60 (Speedsolving Wiki, 2021)), which suggests that there are additional insights to be mined from human strategy beyond just learning focused macro-actions.

5.3.5 Interpretability of Focused Macros

We examined the learned focused macros for several domains and found that in addition to having low effect size, they were also frequently easy to interpret. In 15-Puzzle, one type of macro swapped the blank space with a central tile; another type exchanged three tiles without moving the blank space. In Rubik's cube, one macro (Figure 5.4b) swapped three edge-corner pairs while keeping them connected. In Tower of Hanoi, macros moved stacks of disks at a time from one peg to another. We remark that this is quite similar to the interpretability of the human expert macros in Rubik's cube.

[4] Macro-action sequences are included in Appendix C.4.



(a) Expert 3-corner-swap (b) Learned 3-pair-swap

Figure 5.4: Expert and learned macro-actions (Rubik's cube).

5.3.6 Generalizing to Novel Goal States

Since our macro-generation step is goal-independent, we can reuse previously learned macros to solve problems with novel goal states. To demonstrate this, we generate 100 random goal states for 15-puzzle and Rubik's cube and then solve the puzzles again. In both domains, we find that planning time and solve rate remain effectively unchanged for novel goal states (see Table 5.4).

Domain	Goal Type	Generated States	Solve Rate
15-puzzle	Default	4952.4	1.0
	Random	4780.0	1.0
Rubik's cube	Default	171331.4	1.0
	Random	152503.7	1.0

Table 5.4: Average planning efficiency and solve rate, when reusing previously-discovered focused macros to solve 15-puzzle and Rubik's cube with either the default goal state or new randomly-generated goal states. Novel goal state performance is effectively unchanged.

5.4 Related Work

The concept of building macro-actions to improve planning efficiency is not new. Dawson and Siklossy (1977) considered two-action macros and analyzed domain structure to remove macros that were invalid or that had no effect. Korf's (1985) Macro Problem Solver investigated how to learn macros in problems with decomposable operators and serializable subgoals. Macro-FF (Botea et al., 2005) and MUM (Chrpa et al., 2014) learned macros from training problem instances and later used them to improve the planning efficiency of testing instances. The macros we discover in Sections 5.2 and 5.3 can similarly be reused across problem instances; however, our macro discovery procedure requires neither goal information nor training instances. Our method is perhaps most similar to MARVIN (Coles and Smith, 2007), which used macros to escape plateaus during heuristic search, and CAP (Asai and Fukunaga, 2015), which decomposed planning problems into subgoals and then found macros to achieve those subgoals. In all of the prior approaches, the learned macros were found to be beneficial for planning, but they also required an explicit model of the domain. Our method is more general than these methods, as it is designed to handle the unique challenges of black-box planning

without an explicit model.

Lipovetzky and Geffner (2012) introduced Iterated Width (IW) search, a "blind" planner compatible with black-box simulators, and Lipovetzky et al. (2015) subsequently applied it to planning in Atari video game simulators without known goal states. This work led to the goal-informed Best-First Width Search (BFWS) (Lipovetzky and Geffner, 2017; Francès et al., 2017), which we include in our experimental evaluation. Jinnai and Fukunaga (2017) formalized black-box planning and described a method for pruning primitive actions and short macros to avoid generating duplicate states; however their approach did not incorporate goal information.

Recent work by Agostinelli et al. (2019) investigated how to train black-box planning heuristics with neural networks and dynamic programming by strategically resetting the simulator to states near the goal state. Their approach learned heuristics for several domains, including 15-puzzle and Rubik's cube, that supported fast, near-optimal planning. However, training their neural network requires more than 1000 times the simulation budget of our approach, and results in a heuristic that is only informative for a single goal state, whereas ours works for arbitrary goal states.

5.5 Conclusion

We have described a method of learning focused macro-actions that enables reliable and efficient black-box planning across a variety of planning domains. While our approach is designed to match the assumptions of the goal-count heuristic, we find that it also improves the performance of more sophisticated black-box planners. Moreover, our method is even competitive with a state-of-the-art LAMA planner, despite the latter having access to a declarative description of the problem. It is encouraging to see that many of the learned macro-actions had intuitive, interpretable meaning in the task domain. This suggests that our method may be useful for improving explainability in addition to planning efficiency.

This work employed a two-level hierarchy where macro-actions are composed of primitive actions. One extension to bring this method more in line with human-expert techniques would be incorporating additional levels of action hierarchy (i.e. macros composed of other macros), or macros that permit side-effects to certain unsolved variables, combined with macros to subsequently solve those remaining variables. We leave an exploration of these ideas for future work.

Chapter 6

World-Model Abstractions

> Too much knowledge never makes for simple decisions.
>
> —Frank Herbert

Consider a general-purpose agent embedded within the simplified video game environment of Minecraft. A proper open-scope world model must be detailed enough to express a range of possible tasks, such as collecting resources, building shelter, crafting weapons, cooking food, fighting off enemies—anything the agent has the capability to do. Now suppose that the agent's task is to construct a bed. The agent isn't hungry; there are no enemies around; it's safely enclosed in the fence of Figure 6.1. Given the particular circumstances, information about other tasks—like cooking food or fighting enemies—is completely irrelevant, as are most of the objects in the agent's vicinity, since they aren't components of any bed-crafting recipe.

Superfluous actions and objects in the agent's world model shouldn't make the planning problem any harder,

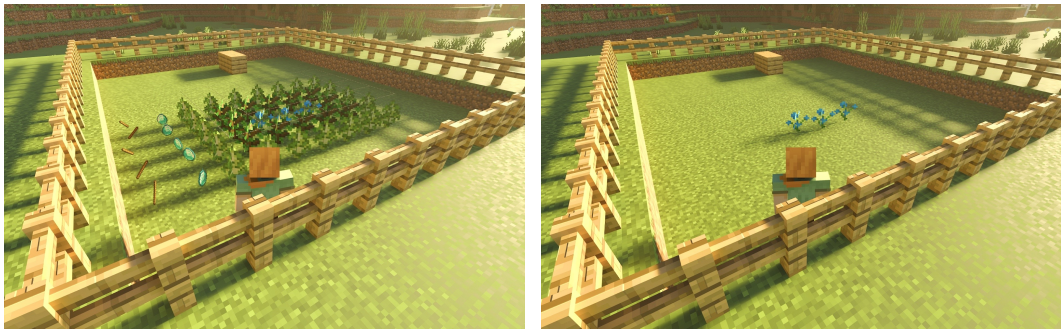


Figure 6.1: (Left) An open-scope Minecraft environment. All of the objects are occasionally important; however, for the specific task of crafting a bed, the planning agent can ignore most of them. (Right) Task scoping removes irrelevant objects and actions prior to planning, reducing planning time by an order of magnitude.

and yet, for most state-of-the-art planners, they do. For the bed-crafting task outlined above, the numeric planner ENHSP (Scala et al., 2020) fails to find an optimal plan, and Fast Downward (Helmert, 2006) fails to even translate a classical version of the same problem. However, with the irrelevant objects and actions removed, both planners can solve the task within just a few minutes.

To enable general-purpose agents to overcome these sorts of challenges automatically, we introduce *task scoping*, a method for reasoning about which information can safely be removed from the agent's open-scope model. We identify three types of task-scoping simplifications that agents can use, after grounding but prior to planning, to remove irrelevant actions and variables. First, agents need only consider actions as relevant if they modify goal variables or preconditions of other relevant actions. Second, agents can identify actions that have identical effects on relevant variables, and merge them. Third, agents can also ignore variables that already match relevant preconditions and goal clauses, unless a relevant action can modify them.

We prove that task scoping preserves all optimal plans and empirically demonstrate that it leads to substantial reductions in search space size and planning time. Applying task scoping to the Minecraft problem of Figure 6.1 allows us to solve the previously intractable problem in under 3 minutes. We also observe significant improvements on a variety of other numeric and classical planning domains. Most importantly, the entire process is automatic, and compatible with off-the-shelf planners, enabling agents to derive their own task-specific simplifications for planning with open-scope models, all without requiring any additional domain knowledge from human experts.

6.1 Background

We adopt the planning formalism of Scala et al. (2016a), but with multivariate enum variables replacing binary variables. We define a planning domain (or model) in terms of the following quantities:

- A finite set of variables $\mathcal{V} = \mathcal{V}_e \cup \mathcal{V}_n$, composed of enums $\mathcal{V}_e$ and numerics $\mathcal{V}_n$. The domain of a variable, $\mathcal{D}(v)$, is finite for enums and the set of reals, $\mathbb{R}$, for numerics.
- A factored state space $\mathcal{S}$, wherein each state s assigns a value $d_i \in \mathcal{D}(v_i)$ to every variable v_i in $\mathcal{V}$.
- A set of grounded operators (i.e. actions) $\mathcal{O}$ where each operator o consists of a cost $c(o) \geq 0$, as well as a precondition $pre(o)$ and an effect $\mathit{eff}(o)$, defined below.

A precondition is a (possibly nested) conjunction of enum and numeric conditions and their negations.[1] An enum condition is an equality relation $v_e = c_e$ between an enum variable v_e and a constant c_e in the domain $\mathcal{D}(v_e)$. When an enum variable is binary, we will write v_e as a shorthand for $v_e = \text{TRUE}$. A numeric condition is a binary relation $\{=, \geq, >\}$ between two numeric expressions, each of which can be either a real-valued constant, a numeric variable, or a binary function $\{+, -, \times, \div\}$ of numeric expressions.

[1] Our implementation is more general and supports any boolean-valued expression that can be expressed using Z3 (De Moura and Bjørner, 2008), but in principle, there are no restrictions on the expressions except that we can identify any associated variables and check effects for equality.

An effect is a list of enum and numeric effects, each to a distinct variable v_i in $\mathcal{V}$. An enum effect is an assignment $v_e := c_e$, where the value c_e is a constant in the variable's domain $\mathcal{D}(v_e)$. A numeric effect is an update to a variable v_n, via an operator $\{:=, +=, -=\}$ and a corresponding numeric expression e_i.

An operator o is applicable in state s if s implies $pre(o)$. Executing o incurs cost $c(o)$ and causes the variable updates described in $\mathit{eff}(o)$, resulting in a new state s'. We reference the variables appearing in $pre(o)$ and $\mathit{eff}(o)$ using the notation $vars(pre(o))$ and $vars(\mathit{eff}(o))$. If an effect assignment to a variable v depends on the value of another variable u, then u is considered a precondition variable of the operator, even if it does not appear in the precondition.

We define a problem instance (or task) of a given domain by adding an initial state $s_0 \in \mathcal{S}$ and a goal condition G consisting of an assignment to some or all of the variables in $\mathcal{V}$. Together, the domain and problem instance induce a grounded planning problem, in which each lifted operator in the domain is instantiated with all possible variable arguments to form a set of grounded operators.[2] A plan is a sequence of successively-applicable grounded operators $[o_1, o_2, ..., o_n]$ from initial state s_0 to final state s_n, such that s_n implies G. An optimal plan is any plan that incurs the minimum cost. Given a model, a task is *solvable* if there exists at least one plan using that model.

This formalism is compatible with both numeric and finite-domain-representation planning and supports a variety of problem encodings, including PDDL 2.1, level 2 (Fox and Long, 2003) and SAS+ (Bäckström, 1992).[3] Our experiments investigate these two planning paradigms separately and assume numeric domains only ever contain binary enums, which is consistent with Scala et al. (2016a). However, in principle our approach is general enough to handle numerics and multivariate enums at the same time, since our algorithm never looks at the domain of any variable.

6.1.1 Defining Task-Relevance

A general-purpose planning agent may be asked solve many tasks during its operational lifetime. If the agent's model contains too few variables or operators, some tasks will not be solvable. To ensure that as many potential tasks as possible are solvable, the agent's model must be open-scope: it must contain more variables and/or operators than are relevant for any one task.

Given a model M and a task t, we say that an operator is *task-relevant* (or simply *relevant*) if it appears in at least one shortest cost-optimal plan to solve t. We say that M is an *open-scope* model with respect to t if it contains operators that are not relevant (henceforth 'irrelevant'). The wider the range of tasks an agent may be asked to solve, the higher the likelihood that most of its operators will be irrelevant for any one of those tasks. These irrelevant operators are the ones we would like the agent to ignore with task scoping.

[2] In this work, we restrict our focus to scoping grounded planning problems, and leave potential scoping-related improvements to the grounding process itself for future work.

[3] For simplicity, we consider the former without conditional effects and the latter without axioms, though neither restriction is required by our approach.

Example 3 (Simplified Minecraft). Consider the following simplified version of the Minecraft domain in Figure 6.1. The agent can collect food, sticks, and stone, which it needs for eating and making an axe. The task is to make an axe from scratch. We use numeric fluents for brevity, but the example does not require them.[a] All operators have unit cost.

- $\mathcal{V} = \{N_{\text{FOOD}},\ N_{\text{STICKS}},\ N_{\text{STONE}} \in \{0,\ 1,\ ...,\ N\};$
  $\text{HUNGRY},\ \text{HAS\_AXE} \in \{\text{TRUE},\ \text{FALSE}\}\}$
- $\mathcal{O} = \{$
  - `hunt` :
    - $pre : \neg(N_{\text{FOOD}} = N) \wedge \neg\text{HUNGRY}$
    - $\mathit{eff} : (N_{\text{FOOD}} \mathrel{+}= 1) \wedge \text{HUNGRY}$
  - `gather` :
    - $pre : \neg(N_{\text{FOOD}} = N) \wedge \text{HUNGRY}$
    - $\mathit{eff} : (N_{\text{FOOD}} \mathrel{+}= 1)$
  - `get_stick` :
    - $pre : \neg(N_{\text{STICKS}} = N)$
    - $\mathit{eff} : (N_{\text{STICKS}} \mathrel{+}= 1)$
  - `get_stone` :
    - $pre : \neg(N_{\text{STONE}} = N)$
    - $\mathit{eff} : (N_{\text{STONE}} \mathrel{+}= 1)$
  - `eat` :
    - $pre : \neg(N_{\text{FOOD}} = 0) \wedge \text{HUNGRY}$
    - $\mathit{eff} : (N_{\text{FOOD}} \mathrel{-}= 1 \wedge \neg\text{HUNGRY})$
  - `make_axe` :
    - $pre : \neg(N_{\text{STICKS}} = 0 \vee N_{\text{STONE}} = 0 \vee \text{HAS\_AXE})$
    - $\mathit{eff} : (N_{\text{STICKS}} \mathrel{-}= 1 \wedge N_{\text{STONE}} \mathrel{-}= 1 \wedge \text{HAS\_AXE})$
  - `wait` :
    - $pre : \neg\text{HUNGRY}$
    - $\mathit{eff} : \text{HUNGRY}\}$
- $s_0 = (0, 0, 0, \text{FALSE}, \text{FALSE})$
- $G = (\neg\text{HUNGRY} \wedge \text{HAS\_AXE})$

For this task, there are two optimal plans: [`get_stick`, `get_stone`, `make_axe`], and [`get_stone`, `get_stick`, `make_axe`]. The operators `hunt`, `gather`, `eat`, and `wait` are irrelevant and can be removed, since none appear in any optimal plan.

[a] Appendix D.3 contains PDDL for this example with $N = 1$.

6.2 Task Scoping

The purpose of task scoping is to identify and remove task-irrelevant variables and operators from the agent's model. This process produces a simplification of the original planning problem aimed at making planning more tractable. However, not all such simplifications preserve optimal plans.

Definition 6. *Given a planning problem P, a* task-scoping *simplification P' of P is one that contains a subset of the variables and operators in P such that all shortest cost-optimal plans in P are still optimal plans of P'.*

In this section, we describe three types of task-scoping simplifications that remove increasing amounts of irrelevant information. We derive the simplifications using variations of Algorithm 2, and prove that each preserves optimal plans.

6.2.1 Backwards Reachability of Variables

The first and simplest task-scoping simplification encodes the notion that agents need not consider actions to be relevant unless they modify goal variables or preconditions of other relevant actions. This version of Algorithm 2 (which we call Algorithm 2-a) omits any of the colored text appearing after the '‖' symbols. It starts by considering only goal variables to be relevant, and performs backwards reachability analysis over variables, considering operators relevant if their effects contain any relevant variables, and then considering the precondition variables of any such operators to be relevant. The process repeats until no new variables are deemed relevant. The Fast Downward Planning System performs an equivalent simplification during its knowledge compilation process (Helmert, 2006).

In the example of Section 6.1.1, this process would work proceed as follows:

HAS_AXE	→	`make_axe`	→	N_{STICKS}	→	`get_stick`
			→	N_{STONE}	→	`get_stone`
HUNGRY	→	`eat`	→	N_{FOOD}	→	`hunt`
					→	`gather`
	→	`wait`				

Eventually all variables and operators would be marked as relevant. Had ¬HUNGRY not appeared in the goal, the algorithm would not consider the bottom chains relevant, and those items would be removed. In Section 6.2.3 we will upgrade the algorithm to recognize that ¬HUNGRY is satisfied by the initial state and not modified by the top operators; therefore, the bottom chains can still be removed.

Algorithm 2 TASK SCOPING

Input: $\langle \mathcal{V}, \mathcal{O}, s_0, G \rangle$
Output: $\langle \mathcal{V}' \subseteq \mathcal{V}, \mathcal{K}' \subseteq \mathcal{V}, \mathcal{O}' \subseteq \mathcal{O} \rangle$

1: $V_0 \leftarrow \{\text{DUMMY\_GOAL\_VAR}\}$ ▷ relevant vars
2: $O_0 \leftarrow \{\texttt{dummy\_goal\_operator}(G)\}$ ▷ relevant ops
3: **repeat**
4: $\quad \overline{O}_i \leftarrow O_{i-1} \parallel \text{MERGESAMEEFFECTS}(O_{i-1}, V_{i-1})$
5: $\quad E_i \leftarrow \varnothing \parallel \{\text{variables} \in \mathit{eff}(O_{i-1})\}$
6: $\quad L_i \leftarrow \varnothing \parallel s_0[\mathcal{V} \setminus E_i]$
7: $\quad C_i \leftarrow \{\text{clauses in } \mathit{pre}(\overline{O}_i) \text{ not implied by } L_i\}$
8: $\quad K_i \leftarrow \{\text{vars in clauses of } \mathit{pre}(\overline{O}_i)\} \setminus \mathit{vars}(C_i)$ ▷ causally linked variables
9: $\quad V_i \leftarrow V_{i-1} \cup \{v \in \mathit{vars}(c) \; \forall \, c \in C_i\}$
10: $\quad O_i \leftarrow \{o \in \mathcal{O} : \mathit{eff}(o) \cap V_i \neq \varnothing\}$
11: **until** $V_i = V_{i-1}$
12: $\mathcal{V}' \leftarrow V_n \setminus \{\text{DUMMY\_GOAL\_VAR}\}$
13: $\mathcal{K}' \leftarrow K_n \setminus \{\text{DUMMY\_GOAL\_VAR}\}$
14: $\mathcal{O}' \leftarrow O_n \setminus \{\texttt{dummy\_goal\_operator}(G)\}$
15: **return** $\langle \mathcal{V}', \mathcal{K}', \mathcal{O}' \rangle$

Algorithm 3 MERGESAMEEFFECTS

Input: O_i, V_i
c **Output**: $\overline{O}_i$

1: $O_{\text{equiv}} \leftarrow$ Partition O_i based on effects on V_i and cost.
2: $\overline{O}_i \leftarrow$ merge operators in equivalence classes: {
$\quad \mathit{pre}$: take disjunction of preconditions & simplify
$\quad \mathit{eff}$: copy effects on V_i (identical)
$\quad c$: copy cost of component operators (identical)}
3: **return** $\overline{O}_i$

6.2.2 Merging Same-Effect Operators

The second task-scoping simplification reflects the idea that actions with identical effects on relevant variables are interchangeable and can therefore be merged. Algorithm 2-b extends the previous version by adding the MERGESAMEEFFECTS function on line 4, which is detailed in Algorithm 3.

The merging procedure partitions the relevant operators O_i into equivalence classes that have identical costs and effects on relevant variables V_i. Each resulting merged operator removes any effects on non-relevant variables, and its precondition is the disjunction of the original operators (simplified to remove any unnecessary clauses using Z3). As a result, line 7 of Algorithm 2 now produces potentially fewer precondition clauses C_i from which to add relevant variables in line 8. Note that the merged operators never appear in $\mathcal{O}'$, only the non-merged originals.

For example, suppose N_{FOOD} is the only relevant variable (perhaps the task is now to gather food). The operators `hunt` and `gather` both modify N_{FOOD}, so both are marked as relevant. Algorithm 2-b then calls

MERGESAMEEFFECTS and determines that both operators have the same effect on N_{FOOD}, the only relevant variable, and can therefore be merged. After simplifying preconditions, the merge results in the following operator (name added for clarity):

`get_food` :
$pre : \neg(N_{\text{FOOD}} = N)$
$\mathit{eff} : (N_{\text{FOOD}} \mathrel{+}= 1)$

In this example, HUNGRY was not relevant to begin with and does not appear in the merged operator's precondition, so it would remain irrelevant. As a result, the `eat` and `wait` operators, which modify HUNGRY, never become relevant either, and the algorithm will return $\mathcal{O}' = \{\texttt{hunt}, \texttt{gather}\}$.

6.2.3 Causally Linked Irrelevance

The third task-scoping simplification we introduce captures the idea that agents can ignore variables that already match relevant preconditions and goal clauses, unless a relevant action modifies them. This corresponds to the concept of causal links (McAllester and Rosenblitt, 1991). A clause is *causally linked* when (1) it is implied by some state (here we only consider causal links from s_0), (2) it appears in the precondition of a subsequent operator, and (3) it is not modified by any operators in between. In the Minecraft example of Section 6.1.1, this corresponded to the variable HUNGRY. Since the initial state and goal both contained the clause $\neg$HUNGRY and no relevant operator modified it, HUNGRY was *causally linked* and could safely be removed.

The full Algorithm 2 builds on the previous version by additionally identifying clauses L_i (in lines 5-6) that are causally linked with the initial state s_0 and contain no overlap with any variables mentioned in the effects of relevant operators.[4] Note that Algorithm 2 is guaranteed to terminate, since $\mathcal{V}$ is finite and $V_{i-1} \subseteq V_i \subseteq \mathcal{V}$ for every iteration.

6.2.4 Main Theorem

We show in this section that Algorithm 2 produces a simplification of the original planning problem that contains all optimal plans. The proof works by removing unnecessary operators from each plan, and makes use of a Merge Substitution lemma, which ensures that the resulting action sequences are still valid plans.

For any quantity X_i in the algorithm, we will use the subscript X_n to indicate the value X_i has in the final iteration of the algorithm. For example, K_n is the final set of causally linked variables. Additionally, for a set of state variables $Z \subseteq \mathcal{V}$, we use the notation $s[Z]$ to denote the partial state of s with respect to only the variables in Z.

[4] Additional optimizations are possible, such as by considering subsequent states or ignoring variables in C_i when their values do not affect the truth value of the causally linked clauses.

Lemma 6.2.1 (Merge Substitution)**.** *Let o be any operator in O_n, s any state that implies $pre(o)$, and $s' \neq s$ another state. If the partial state of s' with respect to relevant and causally linked variables is the same as in state s (i.e. $s'[V_n \cup K_n] = s[V_n \cup K_n]$), then there exists another operator o', also in O_n (and possibly equal to o), such that: s' implies $pre(o')$; o and o' have the same effect on V_n; neither o nor o' has any effect on K_n; and o and o' have the same cost.*

Proof. Run MERGESAMEEFFECTS(O_n, V_n) to compute $\overline{O}_n$, and find the (potentially merged) operator $\overline{o}$ corresponding to o. Such an operator exists, because MERGESAMEEFFECTS partitions O_n.

First we will show that s' implies $pre(\overline{o})$. By construction, $pre(\overline{o})$ contains only variables in $(V_n \cup K_n)$, and the clauses containing variables in K_n are causally linked. This means that if s implies $pre(\overline{o})$, and $s'[V_n \cup K_n] = s[V_n \cup K_n]$, then s' also implies $pre(\overline{o})$. Since $pre(\overline{o})$ is just the disjunction of the preconditions of its component operators, it must be the case that s' implies the precondition of at least one such component operator $o' \in O_n$.

Since o' and o correspond to the same abstract operator $\overline{o}$, they must have the same effect on V_n and the same cost. Since both operators are in O_n, neither can affect K_n. □

Theorem 3. *Every shortest cost-optimal plan for a given task uses a subset of the operators returned by Algorithm 2.*

Proof. We will show that for any plan π containing operators not in O_n, there exists another plan π' that is shorter, has lesser or equal cost, and only uses operators in O_n.

We will go through the operators of π, from the beginning to the end, and for each operator o, add a corresponding operator to π' as follows:

1. If o affects at least one variable in V_n and can be taken from the current state of the modified plan, keep it; o is in O_n, since it modifies a variable in V_n.
2. If o affects at least one variable in V_n and cannot be taken from the current state of the modified plan, replace it with an operator o' that has the same effects on V_n and can be taken from the current state. Such an operator is guaranteed to exist by Lemma 6.2.1.
3. If o does not affect V_n (and therefore is not in O_n), add a no-op operator to π' with the same cost as o.

Note that in case 3, the no-op operators need not exist in the domain. We could simply ignore o entirely in this case, but this would make the correspondences between the operators and states of π and π' slightly less clear, since the plans would have different lengths. We will delete the no-ops from π' after comparing the plans.

Now we provide an inductive proof over the steps in the plan to show that the following inductive assumption holds.

Inductive assumption: At each step, π and π' have the same partial state on V_n, and π''s partial state on K_n (the set of causally linked variables) is equal to the initial partial state on K_n. That is, $s_i[V_n] = s'_i[V_n]$ and $s'_i[K_n] = s_0[K_n]$.

Inductive base: Empty plans share initial state.

Inductive step: Each of the three cases outlined above preserves the inductive assumption.

- Case 1: the original operator o is applicable and does not change the plan. The resulting partial state $s'_{i+1}[V_n] = s_{i+1}[V_n]$. Since o is in O_n, it does not modify K_n.
- Case 2: this operator replacement is possible due to Lemma 6.2.1 and the inductive assumption. By Lemma 6.2.1, o and o' have the same effect on V_n, no effect on K_n, and equal cost. Furthermore, o' is in O_n.
- Case 3: o does not modify V_n, so $s'_{i+1}[V_n] = s_{i+1}[V_n]$. The no-op does not affect K_n, and has equal cost to o.

The above inductive argument shows that π' is also a valid plan, and has the same cost and length as π. We now delete the no-ops from π', and afterwards, π' is shorter than π, has lesser or equal cost, and uses only operators from O_n.

For any plan π with an operator not in O_n, there exists a shorter plan of equal or lesser cost, containing only operators in O_n. Therefore, all shortest cost-optimal plans use only operators from O_n. □

6.2.5 Discussion

It should be clear that Algorithm 2 does not involve searching through specific states. Rather, it reasons over the variables and operators of the planning problem. Space precludes a thorough complexity analysis, but Algorithm 2's worst-case complexity is dominated by $|\mathcal{V}| \times |\mathcal{O}|$. Since $|\mathcal{V}| \times |\mathcal{O}|$ is generally much smaller than the problem's full state–action space (which may even be infinite), deriving a task-scoping simplification is thus often significantly more efficient than planning over the original problem.[5]

The simplifications in the previous sections are not intended to be an exhaustive list of task-scoping simplifications. More aggressive simplifications are clearly possible, particularly when considering causal links.

6.3 Experimental Evaluation

Algorithm 2 is agnostic to which particular representation is used to express the planning problem. To demonstrate its performance and utility empirically, we customized it to work with both numeric PDDL 2.1

[5]This complexity expression neglects the cost of simplifying preconditions in Algorithm 2, which is reasonable in practice because most preconditions are relatively simple.

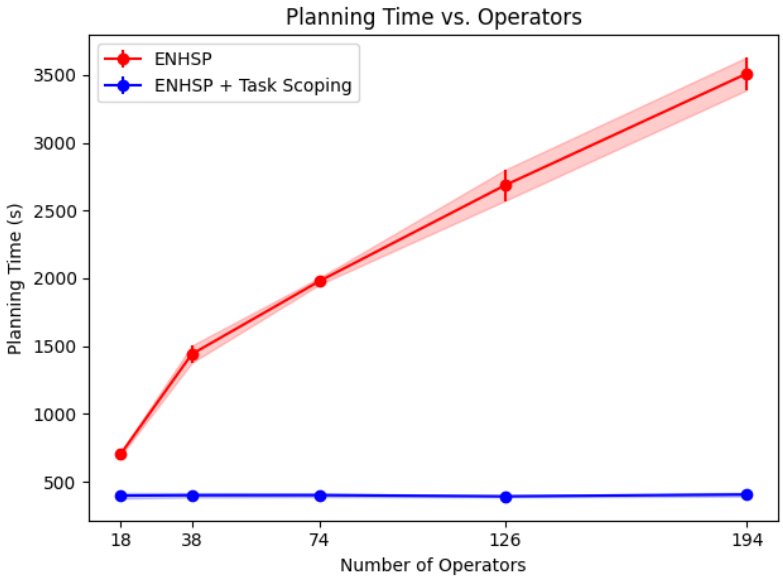


Figure 6.2: Results for the Multi-Switch Continuous Playroom domain. Total planning time with task scoping is essentially constant compared to the baseline, even as the size of the domain grows exponentially. (Total planning time includes scoping time; error bars show standard deviation across 10 independent trials.)

level 2 (Fox and Long, 2003), and SAS+ (Bäckström, 1992; Helmert, 2006). The output of the algorithm consists of relevant variables $\mathcal{V}'$, causally linked variables $\mathcal{K}'$, and relevant operators $\mathcal{O}'$. For SAS+, we simply remove operators outside of $\mathcal{O}'$ and causally linked goal conditions. For PDDL, we remove lifted operators from the domain file whenever they correspond to no grounded operators in $\mathcal{O}'$, and we remove objects from the problem file whenever they correspond to no variables in $\mathcal{V}' \cup \mathcal{K}'$.[6] Since this is a more conservative simplification, it is still a valid task-scoping simplification.

All experiments were conducted on a cluster of 2.90GHz Intel Xeon Platinum 8268 CPUs, using 2 virtual cores and 16GB of RAM per trial, and measurements are averaged across 10 independent trials.

6.3.1 Numeric Domains with ENHSP

We first investigate our approach's performance and utility in numeric domains. In all the following experiments, we run ENHSP (Scala et al., 2020) in optimal mode (`WAStar+hrmax`) with and without task scoping as a pre-processing step. We measure wall-clock time and nodes generated during search to produce a plan.

Multi-Switch Continuous Playroom. In order to study how our approach scales with the size of the input problem in a simplified setting, we implemented the continuous playroom domain from Singh et al. (2004a) in PDDL 2.1. In this domain, the agent can move in cardinal directions within a grid. Its goal is to pick up a ball and throw it at a bell, but this requires first pressing a number of green buttons, which in turn requires

[6] We keep objects corresponding to variables in $\mathcal{K}'$ because these objects may be used to ground operators in $\mathcal{O}'$.

Problem	Operators		State Variables		Evaluations		Scoping	Planning	Total Time (s)	
	Unscoped	Scoped	Unscoped	Scoped	Unscoped	Scoped		Scoped	Unscoped	Scoped
Playroom 1	18	**14**	22	**12**	13.0M	**11.3M**	0.3 ± 0.1	397 ± 20	703 ± 13	$\mathbf{397 \pm 20}$
Playroom 3	38	**14**	48	**12**	14.9M	**11.3M**	0.5 ± 0.0	400 ± 15	1440 ± 63	$\mathbf{401 \pm 15}$
Playroom 5	74	**14**	82	**12**	15.2M	**11.3M**	0.9 ± 0.0	400 ± 13	1981 ± 25	$\mathbf{401 \pm 13}$
Playroom 7	126	**14**	124	**12**	15.2M	**11.3M**	1.5 ± 0.1	391 ± 6	2686 ± 114	$\mathbf{393 \pm 6}$
Playroom 9	194	**14**	174	**12**	15.2M	**11.3M**	2.3 ± 0.1	403 ± 14	3510 ± 121	$\mathbf{406 \pm 14}$
Composite (depot)	1809	**1062**	592	**154**	(> 19.0M)	**488K**	26.6 ± 0.3	84.9 ± 0.9	(> 5.8K)	$\mathbf{111.5 \pm 1.1}$
Composite (driverlog)	1809	**336**	592	**263**	(> 18.6M)	**66K**	22.6 ± 0.1	4.7 ± 0.3	(> 4.3K)	$\mathbf{27.3 \pm 0.4}$
Composite (satellite)	1809	**94**	592	**157**	(> 22.1M)	**92K**	23.4 ± 0.2	3.8 ± 0.4	(> 5.3K)	$\mathbf{27.2 \pm 0.4}$
Composite (zenotravel)	1809	**37**	592	**36**	1.7M	**598**	11.7 ± 0.1	1.1 ± 0.1	363 ± 2	$\mathbf{12.7 \pm 0.1}$
Minecraft (planks)	106	**22**	268	**58**	353	**338**	4.8 ± 0.1	1.3 ± 0.2	$\mathbf{1.7 \pm 0.2}$	6.1 ± 0.3
Minecraft (wool)	106	**18**	268	**58**	1.6M	**6612**	4.1 ± 0.1	1.7 ± 0.2	432.5 ± 21.5	$\mathbf{5.8 \pm 0.2}$
Minecraft (bed)	106	**25**	268	**64**	(> 14.2M)	**1.1M**	5.0 ± 0.1	173.1 ± 8.0	(> 4.3K)	$\mathbf{178.1 \pm 8.0}$

Table 6.1: Numeric planning results. Task scoping removes irrelevant operators and state variables, which leads to significant reductions in evaluations and planning time. Boldface denotes better performance. Standard deviation is across 10 trials. $(> N)$ denotes out-of-memory.

turning on a number of lights. Both the lights and buttons are strewn throughout the grid. In our experiments, we create progressively larger problems with more buttons and lights, yet ensure that all green buttons are turned on in the initial state. This renders the lights causally-linked with the initial state, so all corresponding actions that affect them are irrelevant.

The results, shown in Figure 6.2, demonstrate that task scoping is able to keep planning time relatively constant compared to the baseline, even as the size of the problem grows exponentially. Table 6.1 provides more detail, and shows that even after accounting for the time required to perform task scoping, the scoped planner still finds plans more quickly than the unscoped baseline.

Composite IPC Domains. Task scoping is intended to make *open-scope* planning tractable, but most existing benchmarks are not open-scope: they have been carefully hand-designed to exclude any task-irrelevant information. While a full investigation of how to *learn* such an open-scope model is beyond the scope of this dissertation, it is straightforward to construct an open-scope domain from existing benchmarks: simply combine multiple domains together and attempt to solve a problem from any one of them.

We construct such a model by combining the domain and problem files of the DEPOTS, DRIVERLOG, SATELLITE, and ZENOTRAVEL domains from IPC 2002 (Long and Fox, 2003) into a single composite domain file and 4 different composite problem files. The problem files differ only in their goal condition: each file contains a goal related to a specific sub-domain, and the variables and actions related to the other 3 domains are task-irrelevant.

The results in Table 6.1 indicate that task scoping can dramatically reduce the number of operators, which makes optimal open-scope planning tractable in all four tasks, whereas ENHSP typically runs out of memory without scoping.

Minecraft. To examine the utility of task scoping on a novel open-scope domain of interest, we created a planning domain containing simplified dynamics and several tasks from Minecraft and pictured in Figure 6.1. The domain features interactive items at various locations in the map, which the agent can destroy, place

Problem	Operators		Expansions		Evaluations		Translate	Scoping	Planning Time (s)		Total Time (s)	
	Unscoped	Scoped	Unscoped	Scoped	Unscoped	Scoped			Unscoped	Scoped	Unscoped	Scoped
Driverlog 15	2592	**2112**	1392	**1379**	23K	**21K**	0.5 ± 0.0	3.6 ± 0.2	4.4 ± 0.1	$\mathbf{3.1 \pm 0.1}$	$\mathbf{4.9 \pm 0.2}$	7.2 ± 0.2
Driverlog 16	4890	**3540**	3618	**3087**	87K	**60K**	0.7 ± 0.0	8.3 ± 0.3	19.8 ± 0.8	$\mathbf{8.1 \pm 0.2}$	20.5 ± 0.8	$\mathbf{17.1 \pm 0.5}$
Driverlog 17	6170	**3770**	1058	**985**	29K	**21K**	0.8 ± 0.0	9.6 ± 0.3	22.0 ± 0.9	$\mathbf{7.9 \pm 0.2}$	22.8 ± 1.0	$\mathbf{18.3 \pm 0.5}$
Logistics 15	650	**250**	6395	6395	153K	**118K**	0.3 ± 0.1	0.7 ± 0.1	11.6 ± 0.2	$\mathbf{3.3 \pm 0.1}$	11.9 ± 0.2	$\mathbf{4.2 \pm 0.1}$
Logistics 20	650	**250**	15K	**14K**	381K	**260K**	0.3 ± 0.0	0.7 ± 0.0	26.9 ± 0.3	$\mathbf{5.9 \pm 0.1}$	27.2 ± 0.3	$\mathbf{6.9 \pm 0.2}$
Logistics 25	650	**290**	68K	**67K**	1.7M	**1.3M**	0.3 ± 0.0	0.8 ± 0.0	127.7 ± 1.4	$\mathbf{34.8 \pm 0.4}$	128.0 ± 1.4	$\mathbf{35.8 \pm 0.4}$
Satellite 05	609	**339**	1034	1034	63K	**35K**	0.3 ± 0.1	0.6 ± 0.1	2.0 ± 0.0	$\mathbf{0.6 \pm 0.0}$	2.3 ± 0.1	$\mathbf{1.5 \pm 0.0}$
Satellite 06	582	**362**	5766	**4886**	312K	**166K**	0.3 ± 0.1	0.5 ± 0.1	6.3 ± 0.1	$\mathbf{1.7 \pm 0.0}$	6.6 ± 0.1	$\mathbf{2.5 \pm 0.0}$
Satellite 07	983	**587**	125K	**96K**	10.5M	**4.9M**	0.4 ± 0.1	0.9 ± 0.1	333.1 ± 1.8	$\mathbf{50.9 \pm 0.3}$	333.4 ± 1.8	$\mathbf{52.2 \pm 0.3}$
Zenotravel 10	1155	**1095**	24K	24K	676K	**655K**	0.3 ± 0.0	2.4 ± 0.1	37.7 ± 0.7	$\mathbf{33.2 \pm 0.4}$	38.1 ± 0.7	$\mathbf{35.9 \pm 0.5}$
Zenotravel 12	3375	**3159**	4766	**4735**	222K	**211K**	0.6 ± 0.0	7.4 ± 0.1	45.5 ± 0.1	$\mathbf{39.2 \pm 0.3}$	$\mathbf{46.0 \pm 0.2}$	47.2 ± 0.3
Zenotravel 14	6700	**6200**	6539	6539	599K	**588K**	1.0 ± 0.0	14.2 ± 0.2	232.4 ± 18.5	$\mathbf{193.2 \pm 8.7}$	233.4 ± 18.5	$\mathbf{208.3 \pm 8.6}$

Table 6.2: Classical planning results. Task scoping removes irrelevant operators from every domain, which leads to significant reductions in node expansions, evaluations, and planning time. Boldface denotes better performance. Standard deviation is across 10 trials.

elsewhere, or use to craft different items. Thus, the domain is truly open-scope: it supports a large variety of potential tasks such that most objects and actions are irrelevant to most tasks. Within this domain, we wrote PDDL 2.1 files to express 3 specific tasks: (1) craft wooden planks, (2) dye 3 wool blocks blue, and (3) craft and place a blue bed at a specific location (which requires completing both prior tasks as subgoals). The results show that task scoping is able to recognize and remove a large number of irrelevant operators depending on the task chosen within this domain, as shown in Table 6.1. This dramatically speeds up planning time for the wool-dyeing and bed-making tasks.[7]

6.3.2 Classical Planning with Fast Downward

Having investigated our approach's utility and performance in a variety of numeric domains, we now turn to propositional domains. We are interested in examining whether our approach is able to discover task-irrelevant information beyond what the translator component of the well-known Fast-Downward planning system (Helmert, 2006), and whether removing such irrelevance can substantially improve planning time. To this end, we selected 4 benchmark domains (Logistics, DriverLog, Satellite, Zenotravel) from the optimal track of several previous iterations of the International Planning Competition (IPC) (Long and Fox, 2003; Vallati et al., 2015; Gerevini et al., 2009; Long et al., 2000). Since these domains do not contain any task-irrelevance on their own (Hoffmann et al., 2006), we modified 3 problem files from each domain with initial states and goals set to introduce irrelevance while keeping the domain files fixed. We grounded each problem to SAS+ using FD's translator, ran task scoping on the resulting SAS+ file, then ran the FD planner with the LM-cut heuristic (Helmert and Domshlak, 2009) on this problem. We report number of operators, planning time, and nodes expanded and evaluated during search both with and without task scoping.

The results (see Table 6.2) reveal that task scoping can abstract some of these problems beyond what FD's translator can accomplish alone and lead to a net speedup. Algorithm 2 reduces the number of operators significantly for all 4 domains. This difference was mostly because FD's translator was unable to remove any

[7] We also created a propositional version of this domain. Fast Downward was not even able to complete translation on it; however, after removing irrelevant objects for each problem by hand, planning took just a few seconds.

causally-linked irrelevant variables or operators, though it was able to remove the simpler types of irrelevance discussed in Section 6.2.1.

6.4 Related Work

The Fast Downward Planning System (Helmert, 2006) performs Algorithm 2-a from Section 6.2.1 as part of its knowledge compilation process. This is backwards reachability analysis on what the authors call the *achievability* causal graph. The MERGESAMEEFFECTS extension can be interpreted as computing abstract operators with fewer preconditions, making the causal graph sparser. The causal links extension (lines 5–7 of Algorithm 2) also makes the causal graph sparser by ignoring satisfied clauses of preconditions. The sparser causal graph means that the backwards reachability analysis terminates sooner, with fewer variables marked as potentially relevant.

Another popular method for reducing the size of the search space is the discovery of forward and backward invariants (a.k.a mutex constraints) (Bonet and Geffner, 2001; Edelkamp and Helmert, 1999; Chen et al., 2007; Alcázar and Torralba, 2015). Removing such invariants removes unreachable states or dead-end states, and their associated operators, from the planning problem and has been shown to dramatically improve search (Helmert, 2006). However, removing invariants essentially amounts to removing states and operators that cannot be part of any plan. By contrast, our approach removes states that are very much reachable from both the initial and goal states; removing them does not preserve all plans, but rather all *optimal* plans.

Some recent work removes operators and corresponding states (Fišer et al., 2019; Horčík and Fišer, 2021) that may be part of plans, but can still be safely ignored to preserve at least one optimal plan. This research is based on the central idea that some operators, or transitions (Haslum et al., 2013; Torralba and Kissmann, 2015), in plans may be strictly dominated by others, and thus can be safely removed. Our work can be seen as focusing on efficiently removing a subset of such dominated operators and states. Importantly, these existing approaches depend on problems having a finite state space—they often rely on “factorizing" a problem into smaller problems (Horčík and Fišer, 2021; Torralba and Hoffmann, 2015) and performing potentially expensive operations like symmetry-checking or constraint-satisfaction over these smaller problems. By contrast, our approach can handle infinite state spaces (as long as the number of variables and operators is finite), and Algorithm 2’s complexity does not scale with the size of the state space, but rather with the number of (grounded) variables and operators.

Yet another line of work involves using abstractions to derive heuristics to guide search within the concrete problem (Culberson and Schaeffer, 1998; Nebel et al., 1997; Katz and Domshlak, 2010). Some of these approaches can use richer families of abstractions than Algorithm 2 (for example, Cartesian abstractions). However, such approaches do not directly remove irrelevance from planning tasks, since the resulting abstractions do not necessarily preserve any valid plans. Some of this research (Rovner et al., 2019; Seipp and Helmert, 2018) performs an iterative abstraction refinement similar to our approach, but interleaves planning and abstraction refinement whereas Algorithm 2 does not need to perform planning to refine its simplification.

6.5 Conclusion

Task scoping enables existing domain-independent planners to generalize to a much broader class of *open-scope* planning problems. By carefully removing irrelevant variables and actions from consideration, our algorithm allows planners to overcome the exponential cost of planning with large amounts of irrelevant information. This reduction in problem complexity leads to substantial improvements in planning time, even after accounting for the time spent deriving such simplifications. Moreover, planners that use these simplifications do not suffer any penalty in terms of plan quality, as all optimal plans are guaranteed to be preserved under our algorithm. This work builds on the already impressive legacy of domain-independent planners as general-purpose problem solvers, and represents an important step on the path to realizing truly general decision-making agents.

Chapter 7

Closing Remarks

> I never am really satisfied that I understand anything; because, understand it well as I may, my comprehension can only be an infinitesimal fraction of all I want to understand.
>
> —Ada Lovelace

General-purpose decision making requires agents to be highly adaptible. Their effectiveness depends on their ability to overcome challenges that their designers did not anticipate. Such flexibility demands both a sensorimotor space rich enough to support a wide array of tasks, and the capacity to simplify the inevitable complexity contained therein. This dissertation investigated several ways to build agents that automatically learn useful decision-making abstractions for both reinforcement learning and planning.

In reinforcement learning, we developed a method that learns abstract state representations to deal with noisy, high-dimensional observations. The method overcomes longstanding challenges in representation learning with a practical training objective that provably retains sufficient information to characterize the environment's Markov transition dynamics. The learned representations were human-interpretable and led to state-of-the-art performance across a wide range of applications.

We then expanded our focus to general decision processes where the agent's observations were insufficient and required augmentation with memory. We introduced the λ-discrepancy and proved that it is a reliable indicator of non-Markov observations. We then used this quantity as a training signal to learn history abstractions that resolve partial observability, thereby supporting both the estimation of Markov value functions and the expression of Markov optimal policies.

Next, we explored action abstractions that align with the demands of factored black-box planning. We constructed libraries of skills with focused, interpretable effects, and in so doing, enabled black-box planners

to make substantially better use of their goal-counting-based planning heuristics. This led to dramatically faster planning without any reliance on human-designed action schemas.

Finally, we looked at world-model abstractions that ignore task-irrelevant complexity. We introduced three forms of task-scoping simplifications based on backwards relevance chaining, operator merging, and causal links analysis, and proved that all three preserve optimal plans. We then showed that scoping tasks prior to planning can greatly decrease subsequent planning time while adding virtually no overhead, often improving efficiency by orders of magnitude.

Together, these examples provide compelling evidence for the thesis that by discovering and exploiting the structure in their environment, agents can autonomously build decision-making abstractions that are beneficial for learning and planning.

They also raise interesting and important new questions. How should an agent decide which abstraction method to employ first? For abstractions that rely on having a factored state space, where might such a factorization come from? For tasks involving planning, how does the agent acquire its world model? If the world model is to be learned, how can the agent ensure it is accurate? Can the agent ever guarantee that the environment has been sufficiently explored? Even more fundamentally, which decision making framework is most appropriate for the problem at hand? Seeking answers to these questions is essential if we hope to one day build truly general-purpose agents that are more than just mere reflections of their designers' intelligence.

Appendix

Appendix A

Markov State Abstractions

A.1 Limitations

Our approach enables agents to automatically construct useful, Markov state representations for reinforcement learning from rich observations. Since most reinforcement learning algorithms implicitly assume Markov abstract state representations, and since agents may struggle to learn when that assumption is violated, this work has the potential to benefit a large number of algorithms.

Our training objective is designed for neural networks, which are not guaranteed to converge to a global optimum when trained with stochastic gradient descent. Typically, such training objectives will only approximately satisfy the theoretical conditions they encode. However, this is not a drawback of our method—it applies to any representation learning technique that uses neural networks. Moreover, as neural network optimization techniques improve, our method will converge to a Markov abstraction, whereas other approaches may not. In the meantime, systems in safety-critical domains should ensure that they can cope with non-Markov abstractions without undergoing catastrophic failures.

We have shown experimentally that our method is effective in a variety of domains; however, other problem domains may require additional hyperparameter tuning, which can be expensive. Nevertheless, one benefit of our method is that Markov abstractions can be learned offline, without access to reward information. This means our algorithm could be used, in advance, to learn an abstraction for some problem domain, and then subsequent tasks in that environment (perhaps with different reward functions) could avoid the problem of perceptual abstraction as well as any associated training costs.

A.2 Glossary of Symbols

Here we provide a glossary of the most commonly-used symbols appearing in the rest of the chapter.

Symbol	Description
$M = (X, A, R, T, \gamma)$	Ground MDP
X	Observed (ground) state space
A	Action space
$R(x', a, x)$	Reward function
$T(x'\|a, x)$	Transition model
γ	Discount factor
$R^{(k)}(x', a, x, \{\cdots\}_{i=1}^{k})$	Reward function conditioned on k steps of additional history
$T^{(k)}(x'\|a, x, \{\cdots\}_{i=1}^{k})$	Transition model conditioned on k steps of additional history
S	Unobserved (latent) state space, assumed for block MDPs
$\sigma : S \to \Pr(X)$	Sensor function for producing observed states
$\sigma^{-1} : X \to S$	Hypothetical "inverse-sensor" function, assumed for block MDPs
$\phi : X \to Z$	Abstraction function
$w(x)$	Fixed ground-state weighting function for constructing an abstract MDP (Li et al., 2006)
$B_{\phi,t}^{\pi}(x\|z_t)$	Belief distribution for assigning policy- and time-dependent weights to ground states
$B_{\phi,t}^{\pi(k)}(x\|z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k})$	Belief distribution conditioned on k steps of history
$M_\phi = (Z, A, R_{\phi,t}^{\pi}, T_{\phi,t}^{\pi}, \gamma)$	Abstract decision process (possibly non-Markov)
Z	Abstract state space
$R_{\phi,t}^{\pi}(z', a, z)$	Abstract reward function
$T_{\phi,t}^{\pi}(z'\|a, z)$	Abstract transition model
$R_{\phi,t}^{\pi(k)}(z', a, z, \{\cdots\}_{i=1}^{k})$	Abstract reward function conditioned on k steps of additional history
$T_{\phi,t}^{\pi(k)}(z'\|a, z, \{\cdots\}_{i=1}^{k})$	Abstract transition model conditioned on k steps of additional history
$\pi(a\|x)$	A policy
π^*	An optimal policy
Π_C	An arbitrary policy class
Π_ϕ	The class of abstract policies induced by abstraction ϕ
$V^\pi : X \to \mathbb{R}$	The value function induced by π
$P_t^{\pi}(x)$	Ground-state visitation distribution
$P_t^{\pi}(x'\|x)$	Expected next-state dynamics model
$I_t^{\pi}(a\|x', x)$	Inverse dynamics model
$P_{\phi,t}^{\pi}(z)$	Abstract-state visitation distribution
$P_{\phi,t}^{\pi}(z'\|z)$	Abstract expected next-state dynamics model
$I_{\phi,t}^{\pi}(a\|z', z)$	Abstract inverse dynamics model
$P_t^{\pi}(x, a\|x')$	Backwards dynamics model, used for kinematic inseparability

Table A.1: Glossary of symbols

A.3 General-Policy Definitions

A.3.1 General-Policy Definitions

The definitions of $T^{\pi}_{\phi,t}$ and $R^{\pi}_{\phi,t}$ in Section 3.2 only apply when the policy π is a member of Π_ϕ. Here we derive more general definitions that are valid for arbitrary policies, not just those in Π_ϕ. For the special case where $\pi \in \Pi_\phi$, these definitions are equivalent to equations (3.3) and (3.4).

Abstract transition probabilities.

$$\begin{aligned}
&\Pr(z'|a,z) \\
&= \sum_{x' \in X} \Pr(z'|x',a,z) \Pr(x'|a,z) \\
&= \sum_{x' \in X : \phi(x') = z'} \Pr(x'|a,z) \\
&= \sum_{x' \in X : \phi(x') = z'} \sum_{x \in X} \Pr(x'|a,x,z) \Pr(x|a,z) \\
&= \sum_{x' \in X : \phi(x') = z'} \sum_{x \in X} \Pr(x'|a,x,z) \frac{\Pr(a|x,z) \Pr(x|z)}{\sum_{\tilde{x} \in X} \Pr(a|\tilde{x},z) \Pr(\tilde{x}_t|z_t)} \\
T^{\pi}_{\phi,t}(z'|a,z) &:= \sum_{x' \in X : \phi(x') = z'} \sum_{x \in X : \phi(x) = z} T(x'|a,x) \frac{\pi_t(a|x) B^{\pi}_{\phi,t}(x|z)}{\sum_{\tilde{x} \in X} \pi_t(a|\tilde{x}) B^{\pi}_{\phi,t}(\tilde{x}|z)}
\end{aligned}$$

Abstract rewards.

$$\begin{aligned}
&\sum_{r \in R} r \Pr(r|z',a,z) \\
&= \sum_{x' \in X} \sum_{x \in X} \sum_{r \in R} r \Pr(r,x',x|z',a,z) \\
&= \sum_{x' \in X} \sum_{x \in X} \sum_{r \in R} r \Pr(r|x',z',a,x,z) \Pr(x',x|z',a,z) \\
&= \sum_{x' \in X} \sum_{x \in X} R(x',a,x) \frac{\Pr(z'|x',a,x,z) \Pr(x',x|a,z)}{\Pr(z'|a,z)} \\
&= \sum_{x' \in X : \phi(x') = z'} \sum_{x \in X} R(x',a,x) \frac{\Pr(x'|a,x,z) \Pr(x|a,z)}{\Pr(z'|a,z)} \\
&= \sum_{x' \in X : \phi(x') = z'} \sum_{x \in X} R(x',a,x) \frac{\Pr(x'|a,x)}{\Pr(z'|a,z)} \frac{\Pr(a|x) \Pr(x|z)}{\sum_{\tilde{x} \in X} \Pr(a|\tilde{x}) \Pr(\tilde{x}|z)} \\
R^{\pi}_{\phi,t}(z',a,z) &:= \sum_{x' \in X : \phi(x') = z'} \sum_{x \in X : \phi(x) = z} R(x',a,x) \frac{T(x'|a,x)\, \pi_t(a|x) B^{\pi}_{\phi,t}(x|z)}{T^{\pi}_{\phi,t}(z'|a,z) \sum_{\tilde{x} \in X} \pi_t(a|\tilde{x}) B^{\pi}_{\phi,t}(\tilde{x}|z)}
\end{aligned}$$

A.4 Proofs

Here we provide proofs of Theorem 1 and its corollary, which state that the Inverse Model and Density Ratio conditions are sufficient for ϕ and M_ϕ to be Markov. Then, to complement the counterexample from Section 3.2.2, we also present and prove a second theorem which states that, given the Inverse Model condition, the Density Ratio condition is in fact *necessary* for a Markov abstraction.

A.4.1 Main Theorem

The proof of Theorem 1 makes use of two lemmas: Lemma A.4.1, that equal k-step and $(k-1)$-step belief distributions imply equal k-step and $(k-1)$-step transition models and reward functions, and Lemma A.4.2, that equal k-step and $(k-1)$-step belief distributions imply equal $(k+1)$-step and k-step belief distributions. Since the lemmas apply for any arbitrary policy, we use the general-policy definitions from Appendix A.3.

Lemma A.4.1. *Given an MDP M, abstraction ϕ, policy π, initial state distribution P_0, and any $k \geq 1$, if $B^{\pi(k)}_{\phi,t}(x_t|z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k}) = B^{\pi(k-1)}_{\phi,t}(x_t|z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k-1})$, then for all $a_t \in A$ and $z_{t+1} \in Z$:*

$$\begin{aligned} & T^{\pi(k)}_{\phi,t}(z_{t+1}|\{a_{t-i}, z_{t-i}\}_{i=0}^{k}) = T^{\pi(k-1)}_{\phi,t}(z_{t+1}|\{a_{t-i}, z_{t-i}\}_{i=0}^{k-1}) \\ \cap \quad & R^{\pi(k)}_{\phi,t}(z_{t+1}, \{a_{t-i}, z_{t-i}\}_{i=0}^{k}) = R^{\pi(k-1)}_{\phi,t}(z_{t+1}, \{a_{t-i}, z_{t-i}\}_{i=0}^{k-1}). \end{aligned}$$

In the proof below, we start with $B^{\pi(k)}_{\phi,t} = B^{\pi(k-1)}_{\phi,t}$, and repeatedly multiply or divide both sides by the same quantity, or take the same summations of both sides, to obtain $T^{\pi(k)}_{\phi,t} = T^{\pi(k-1)}_{\phi,t}$, then apply the same process again, making use of the fact that $T^{\pi(k)}_{\phi,t} = T^{\pi(k-1)}_{\phi,t}$, to obtain $R^{\pi(k)}_{\phi,t} = R^{\pi(k-1)}_{\phi,t}$.

Proof:

$$B^{\pi(k)}_{\phi,t}\left(x_t \middle| z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k}\right) = B^{\pi(k-1)}_{\phi,t}\left(x_t \middle| z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k-1}\right).$$

Let $a_t \in A$ be any action.

$$\begin{aligned} \Rightarrow \quad & \pi_t(a_t|x_t) B^{\pi(k)}_{\phi,t}\left(x_t \middle| z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k}\right) = \pi_t(a_t|x_t) B^{\pi(k-1)}_{\phi,t}\left(x_t \middle| z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k-1}\right) \\ \Rightarrow \quad & \frac{\pi_t(a_t|x_t) B^{\pi(k)}_{\phi,t}\left(x_t \middle| z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k}\right)}{\sum_{\tilde{x}_t \in X} \pi_t(a_t|\tilde{x}_t) B^{\pi(k)}_{\phi,t}\left(\tilde{x}_t \middle| z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k}\right)} = \frac{\pi_t(a_t|x_t) B^{\pi(k-1)}_{\phi,t}\left(x_t \middle| z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k-1}\right)}{\sum_{\tilde{x}_t \in X} \pi_t(a_t|\tilde{x}_t) B^{\pi(k-1)}_{\phi,t}\left(\tilde{x}_t \middle| z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k-1}\right)}. \end{aligned} \tag{A.1}$$

Let

$$C^{\pi(k)}_{\phi,t} := \frac{\pi_t(a_t|x_t) B^{\pi(k)}_{\phi,t}\left(x_t|z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k}\right)}{\sum_{\tilde{x}_t \in X} \pi_t(a_t|\tilde{x}_t) B^{\pi(k)}_{\phi,t}\left(\tilde{x}_t|z_t, \{a_{t-i}, z_{t-i}\}_{i=1}^{k}\right)} \tag{A.2}$$

Combining (A.1) and (A.2), we obtain:

$$C^{\pi(k)}_{\phi,t} = C^{\pi(k-1)}_{\phi,t} \tag{A.3}$$

$$\Rightarrow \quad \sum_{x_t \in z_t} \sum_{x_{t+1} \in z_{t+1}} T\left(x_{t+1} \mid a_t, x_t\right) C^{\pi(k)}_{\phi,t} = \sum_{x_t \in z_t} \sum_{x_{t+1} \in z_{t+1}} T\left(x_{t+1} \mid a_t, x_t\right) C^{\pi(k-1)}_{\phi,t}$$

$$\Leftrightarrow \quad T^{\pi(k)}_{\phi,t}\left(z_{t+1} \mid \{a_{t-i}, z_{t-i}\}_{i=0}^{k}\right) = T^{\pi(k-1)}_{\phi,t}\left(z_{t+1} \mid \{a_{t-i}, z_{t-i}\}_{i=0}^{k-1}\right). \tag{A.4}$$

Additionally, we can combine (A.3) and (A.4) and apply the same approach for rewards:

$$
\begin{aligned}
& C_{\phi,t}^{\pi(k)} = C_{\phi,t}^{\pi(k-1)} \\
\Rightarrow \quad & \frac{C_{\phi,t}^{\pi(k)}}{T_{\phi,t}^{\pi(k)}(z_{t+1}|\{a_{t-i},z_{t-i}\}_{i=0}^{k})} = \frac{C_{\phi,t}^{\pi(k-1)}}{T_{\phi,t}^{\pi(k-1)}(z_{t+1}|\{a_{t-i},z_{t-i}\}_{i=0}^{k-1})} \\
\Rightarrow \quad & \frac{T(x_{t+1}|a_t,z_t)C_{\phi,t}^{\pi(k)}}{T_{\phi,t}^{\pi(k)}(z_{t+1}|\{a_{t-i},z_{t-i}\}_{i=0}^{k})} = \frac{T(x_{t+1}|a_t,z_t)C_{\phi,t}^{\pi(k-1)}}{T_{\phi,t}^{\pi(k-1)}(z_{t+1}|\{a_{t-i},z_{t-i}\}_{i=0}^{k-1})} \\
\Rightarrow \quad & R_{\phi,t}^{\pi(k)}\left(z_{t+1},\{a_{t-i},z_{t-i}\}_{i=0}^{k}\right) = R_{\phi,t}^{\pi(k-1)}\left(z_{t+1},\{a_{t-i},z_{t-i}\}_{i=0}^{k-1}\right). \qquad \text{(A.5)}
\end{aligned}
$$

□

Lemma A.4.2. *Given an MDP M, abstraction ϕ, policy π, and initial state distribution P_0, if it holds that $B_{\phi,t}^{\pi(k)}(x_t|z_t,\{a_{t-i},z_{t-i}\}_{i=1}^{k}) = B_{\phi,t}^{\pi(k-1)}(x_t|z_t,\{a_{t-i},z_{t-i}\}_{i=1}^{k-1})$, for all $t \geq k$, $z_t \in Z$, and $x_t \in X$ such that $\phi(x_t) = z_t$, then for all $z_{t+1} \in Z$, $x_{t+1} \in X : \phi(x_{t+1}) = z_{t+1}$,*

$$
B_{\phi,t}^{\pi(k+1)}\left(x_{t+1}\middle|z_{t+1},\{a_{t-i},z_{t-i}\}_{i=0}^{k}\right) = B_{\phi,t}^{\pi(k)}\left(x_{t+1}\middle|z_{t+1},\{a_{t-i},z_{t-i}\}_{i=0}^{k-1}\right).
$$

To prove this lemma, we invoke Lemma A.4.1 to obtain $T_{\phi,t}^{\pi(k)} = T_{\phi,t}^{\pi(k-1)}$, and then follow the same approach as before, performing operations to both sides until we achieve the desired result.

Proof:

Let $T_{\phi,t}^{\pi(k)}$ be defined via (3.3) and (3.2). Applying Lemma A.4.1 to the premise gives:

$$
T_{\phi,t}^{\pi(k)}(z_{t+1}|\{a_{t-i},z_{t-i}\}_{i=0}^{k}) = T_{\phi,t}^{\pi(k-1)}(z_{t+1}|\{a_{t-i},z_{t-i}\}_{i=0}^{k-1}).
$$

Returning to the premise, we have:

$$
\begin{aligned}
& B_{\phi,t}^{\pi(k)}\left(x_t\middle|z_t,\{a_{t-i},z_{t-i}\}_{i=1}^{k}\right) = B_{\phi,t}^{\pi(k-1)}\left(x_t\middle|z_t,\{a_{t-i},z_{t-i}\}_{i=1}^{k-1}\right) \\
\Rightarrow \quad & \frac{B_{\phi,t}^{\pi(k)}\left(x_t\middle|z_t,\{a_{t-i},z_{t-i}\}_{i=1}^{k}\right)}{T_{\phi,t}^{\pi(k)}\left(z_{t+1}\middle|\{a_{t-i},z_{t-i}\}_{i=0}^{k}\right)} = \frac{B_{\phi,t}^{\pi(k-1)}\left(x_t\middle|z_t,\{a_{t-i},z_{t-i}\}_{i=1}^{k-1}\right)}{T_{\phi,t}^{\pi(k-1)}\left(z_{t+1}\middle|\{a_{t-i},z_{t-i}\}_{i=0}^{k-1}\right)} \\
\Rightarrow \quad & \sum_{\substack{x_t\in X:\\ \phi(x_t)=z_t}} \frac{T(x_{t+1}|a_t,x_t)B_{\phi,t}^{\pi(k)}\left(x_t\middle|z_t,\{a_{t-i},z_{t-i}\}_{i=1}^{k}\right)}{T_{\phi,t}^{\pi(k)}\left(z_{t+1}\middle|\{a_{t-i},z_{t-i}\}_{i=0}^{k}\right)} = \sum_{\substack{x_t\in X:\\ \phi(x_t)=z_t}} \frac{T(x_{t+1}|a_t,x_t)B_{\phi,t}^{\pi(k-1)}\left(x_t\middle|z_t,\{a_{t-i},z_{t-i}\}_{i=1}^{k-1}\right)}{T_{\phi,t}^{\pi(k-1)}\left(z_{t+1}\middle|\{a_{t-i},z_{t-i}\}_{i=0}^{k-1}\right)} \\
\Rightarrow \quad & B_{\phi,t}^{\pi(k+1)}\left(x_{t+1}\middle|z_{t+1},\{a_{t-i},z_{t-i}\}_{i=0}^{k}\right) = B_{\phi,t}^{\pi(k)}\left(x_{t+1}\middle|z_{t+1},\{a_{t-i},z_{t-i}\}_{i=0}^{k-1}\right).
\end{aligned}
$$

□

We now summarize the proof of the main theorem. We begin by showing that the belief distributions $B_{\phi,t}^{\pi(k)}$ and $B_{\phi,t}^{\pi(k-1)}$ must be equal for $k = 1$, and use Lemma A.4.1 to prove the base case of the theorem. Then we use Lemma A.4.2 to prove that the theorem holds in general.

Proof of Theorem 1:

Base case. For each $\pi \in \Pi_\phi$, let $B^\pi_{\phi,t}$ be defined via (3.1). Then, starting from the Density Ratio condition, for any $z_{t+1}, z_t \in Z$, and $x_{t+1} \in X$ such that $\phi(x_{t+1}) = z_{t+1}$, and any action $a_{t-1} \in A$:

$$
\begin{aligned}
& \frac{P^\pi_{\phi,t}(z_t|z_{t-1})}{P^\pi_{\phi,t}(z_t)} = \frac{P^\pi_t(x_t|z_{t-1})}{P^\pi_t(x_t)} \\
\Rightarrow \quad & \frac{P^\pi_{\phi,t}(z_t|z_{t-1})}{P^\pi_{\phi,t}(z_t)} = \sum_{x_{t-1}\in X} \frac{P^\pi_t(x_t|x_{t-1})}{P^\pi_t(x_t)} B^\pi_{\phi,t}(x_{t-1}|z_{t-1}) \\
\Rightarrow \quad & \frac{P^\pi_t(x_t)}{P^\pi_{\phi,t}(z_t)} = \sum_{x_{t-1}\in X} \frac{P^\pi_t(x_t|x_{t-1}) B^\pi_{\phi,t}(x_{t-1}|z_{t-1})}{P^\pi_{\phi,t}(z_t|z_{t-1})} \cdot \frac{I^\pi_t(a_{t-1}|x_t,x_{t-1})\pi_{\phi,t}(a_{t-1}|z_{t-1})}{I^\pi_{\phi,t}(a_{t-1}|z_t,z_{t-1})\pi_t(a_{t-1}|x_{t-1})} \\
\Rightarrow \quad & \frac{\mathbb{1}[\phi(x_t) = z_t]\, P^\pi_t(x_t)}{\sum_{\tilde{x}_t \in X} P^\pi_t(\tilde{x}_t)} = \mathbb{1}[\phi(x_t) = z_t] \sum_{x_{t-1}\in X} \frac{T(x_t|a_{t-1},x_{t-1}) B^\pi_{\phi,t}(x_{t-1}|z_{t-1})}{T^\pi_{\phi,t}(z_t|a_{t-1},z_{t-1})} \\
\Rightarrow \quad & B^\pi_{\phi,t}(x_t|z_t) = \frac{\mathbb{1}[\phi(x_t) = z_t] \sum_{x_{t-1}\in X} T_\phi(x_t|a_{t-1},x_{t-1}) B^\pi_{\phi,t}(x_{t-1}|z_{t-1})}{\sum_{\tilde{x}_t \in z_t} \sum_{\tilde{x}_{t-1}\in z_{t-1}} T(\tilde{x}_t|a_{t-1},\tilde{x}_{t-1}) B^\pi_{\phi,t}(\tilde{x}_{t-1}|z_{t-1})} \\
\Rightarrow \quad & B^{\pi(0)}_{\phi,t}(x_t|z_t) = B^{\pi(1)}_{\phi,t}(x_t|z_t, a_{t-1}, z_{t-1})
\end{aligned} \tag{A.6}
$$

Here (A.6) satisfies the conditions of Lemma A.4.1 (with $k = 1$), therefore, for all $a_t \in A$:

$$
\begin{aligned}
& T^{\pi(0)}_{\phi,t}(z_{t+1}|a_t, z_t) = T^{\pi(1)}_{\phi,t}(z_{t+1}|a_t, z_t, a_{t-1}, z_{t-1}) \\
\text{and} \quad & R^{\pi(0)}_{\phi,t}(z_{t+1}, a_t, z_t) = R^{\pi(1)}_{\phi,t}(z_{t+1}, a_t, z_t, a_{t-1}, z_{t-1})
\end{aligned}
$$

This proves the theorem for $k = 1$.

Induction on k. Note that (A.6) also allows us to apply Lemma A.4.2. Therefore, by induction on k:

$$
B^{\pi(k+1)}_{\phi,t}\left(x_{t+1}\middle|z_{t+1}, \{a_{t-i}, z_{t-i}\}_{i=0}^k\right) = B^\pi_{\phi,t}(x_{t+1}|z_{t+1}) \quad \forall\, k \geq 1 \tag{A.7}
$$

□

When (A.7) holds, we informally say that the belief distribution is Markov.

Proof of Corollary 1.1: Follows directly from Definition 2 and Lemma A.4.1 via induction on k. □

This first corollary says that a Markov abstraction implies a Markov abstract state representation. The next one says that, if a belief distribution is non-Markov over some horizon n, it must also be non-Markov when conditioning on a single additional timestep.

Corollary 3.1. *If there exists some $n \geq 1$ such that $B^{\pi(n)}_{\phi,t} \neq B^\pi_{\phi,t}$, then $B^{\pi(1)}_{\phi,t} \neq B^\pi_{\phi,t}$.*

Proof: Suppose such an n exists, and assume for the sake of contradiction that $B^{\pi(1)}_{\phi,t} = B^\pi_{\phi,t}$. Then by Lemma A.4.2, $B^{\pi(k)}_{\phi,t} = B^\pi_{\phi,t}$ for all $k \geq 1$. However this is impossible, since we know there exists some $n \geq 1$ such that $B^{\pi(n)}_{\phi,t} \neq B^\pi_{\phi,t}$. Therefore $B^{\pi(1)}_{\phi,t} \neq B^\pi_{\phi,t}$. □

A.4.2 Inverse Model Implies Density Ratio

As discussed in Section 3.2.2, the Inverse Model condition is not sufficient to ensure a Markov abstraction. In fact, what is missing is precisely the Density Ratio condition. Theorem 1 already states that, given the Inverse Model condition, the Density Ratio condition is sufficient for an abstraction to be Markov over its policy class; the following theorem states that it is also necessary.

Theorem 4. *If $\phi : X \to Z$ is a Markov abstraction of MDP $M = (X, A, R, T, \gamma)$ for any policy in the policy class Π_ϕ, and the Inverse Model condition of Theorem 1 holds for every timestep t, then the Density Ratio condition also holds for every timestep t.*

Proof:

Since ϕ is a Markov abstraction, equation (A.7) holds for any $k \geq 1$. Fixing $k = 1$, we obtain:

$$\begin{aligned}
B^{\pi(1)}_{\phi,t}(x'|z',a,z) &= B^{\pi(0)}_{\phi,t}(x'|z') \\
\frac{\mathbb{1}[\phi(x') = z'] \sum_{\tilde{x} \in X} T(x'|a,\tilde{x}) B^{\pi}_{\phi,t}(\tilde{x}|z)}{\sum_{\tilde{x}' \in X:\phi(x')=z'} \sum_{\tilde{x} \in X:\phi(x)=z} T(\tilde{x}'|a,\tilde{x}) B^{\pi}_{\phi,t}(\tilde{x}|z)} &= \frac{\mathbb{1}[\phi(x') = z'] P^{\pi}_t(x')}{\sum_{\tilde{x}' \in X:\phi(\tilde{x}')=z'} P^{\pi}_t(\tilde{x}')} \\
\frac{\sum_{\tilde{x} \in X} T(x'|a,\tilde{x}) B^{\pi}_{\phi,t}(\tilde{x}|z)}{T^{\pi}_{\phi,t}(z'|a,z)} &= \frac{P^{\pi}_t(x')}{P^{\pi}_{\phi,t}(z')} \\
\frac{\sum_{\tilde{x} \in X} T(x'|a,\tilde{x}) B^{\pi}_{\phi,t}(\tilde{x}|z)}{P^{\pi}_t(x')} &= \frac{T^{\pi}_{\phi,t}(z'|a,z)}{P^{\pi}_{\phi,t}(z')} \\
\sum_{\tilde{x} \in X} \frac{I^{\pi}_t(a|x',\tilde{x}) P^{\pi}_t(x'|\tilde{x}) B^{\pi}_{\phi,t}(\tilde{x}|z)}{P^{\pi}_t(x')\pi(a|\tilde{x})} &= \frac{I^{\pi}_{\phi,t}(a|z',z) P^{\pi}_{\phi,t}(z'|z)}{P^{\pi}_{\phi,t}(z')\pi(a|z)} \\
\frac{\sum_{\tilde{x} \in X} I^{\pi}_t(a|x',\tilde{x}) P^{\pi}_t(x'|\tilde{x}) B^{\pi}_{\phi,t}(\tilde{x}|z)}{P^{\pi}_t(x')} &= \frac{I^{\pi}_{\phi,t}(a|z',z) P^{\pi}_{\phi,t}(z'|z)}{P^{\pi}_{\phi,t}(z')}
\end{aligned} \tag{A.8}$$

Here we apply the Inverse Model condition, namely that $I^{\pi}_t(a|x',x) = I^{\pi}_{\phi,t}(a|z',z)$ for all $z, z' \in Z$; $x, x' \in X$, such that $\phi(x') = z'$ and $\phi(x) = z$.

$$\begin{aligned}
\frac{\sum_{\tilde{x} \in X} P^{\pi}_t(x'|\tilde{x}) B^{\pi}_{\phi,t}(\tilde{x}|z)}{P^{\pi}_t(x')} &= \frac{P^{\pi}_{\phi,t}(z'|z)}{P^{\pi}_{\phi,t}(z')} \\
\frac{P^{\pi}_t(x'|z)}{P^{\pi}_t(x')} &= \frac{P^{\pi}_{\phi,t}(z'|z)}{P^{\pi}_{\phi,t}(z')}
\end{aligned} \tag{Density Ratio}$$

□

It may appear that Theorems 1 and 4 together imply that the Inverse Model and Density Ratio conditions are necessary and sufficient for an abstraction to be Markov over its policy class; however, this is not quite true. Both conditions, taken together, are sufficient for an abstraction to be Markov, and, *given the Inverse Model condition*, the Density Ratio condition is necessary. Examining equation (A.8), we see that, had we instead assumed the Density Ratio condition for Theorem 4 (rather than the Inverse Model condition), we would not recover $I^{\pi}_t(a|x',x) = I^{\pi}_{\phi,t}(a|z',z)$, but rather $\sum_{\tilde{x} \in X} I^{\pi}_t(a|x',\tilde{x}) B^{\pi}_{\phi,t}(\tilde{x}|z) = I^{\pi}_{\phi,t}(a|z',z)$. That is, the Inverse Model condition would only be guaranteed to hold in expectation, but not for arbitrary $x \in X$.

A.5 Derivation of Density Ratio Objective

Our Density Ratio objective in Section 3.3 is based on the following derivation, adapted from Tiao (2017).

Suppose we have a dataset consisting of samples $X_c = \{{x'_c}^{(i)}\}_{i=1}^{n_c}$ drawn from conditional distribution $\Pr(x'|x)$, and samples $X_m = \{{x'_m}^{(j)}\}_{j=1}^{n_m}$ drawn from marginal distribution $\Pr(x')$. We assign label $y = 1$ to samples from X_c and $y = 0$ to samples from X_m, and our goal is to predict the label associated with each sample. To construct an estimator, we rename the two distributions $p(x'|y = 1) := \Pr(x'|z)$ and $p(x'|y = 0) := \Pr(x')$ and rewrite the density ratio $\delta(x') := \frac{\Pr(x'|x)}{\Pr(x')}$ as follows:

$$\delta(x') = \frac{p(x'|y=1)}{p(x'|y=0)} = \frac{p(y=1|x')p(x')}{p(y=1)} \frac{p(y=0)}{p(y=0|x')p(x')} = \frac{n_m}{(n_m+n_c)} \frac{(n_m+n_c)}{n_c} \frac{p(y=1|x')}{p(y=0|x')} = \frac{n_m}{n_c} \frac{p(y=1|x')}{1-p(y=1|x')}. \quad \text{(A.9)}$$

When $n_c = n_m = N$, which is the case for our implementation, the leading fraction can be ignored. To estimate $\delta(x')$, we can simply train a classifier $g(x', x; \theta_g)$ to approximate $p(y = 1|x')$ and then substitute g for $p(y = 1|x')$ in (A.9).

However, we need not estimate $\delta(x')$ to satisfy the Density Ratio condition; we need only ensure $\delta_\phi(z') = \mathbb{E}_{B_\phi}[\delta(x')]$. We therefore repeat the derivation for abstract states, this time using q as our renamed distribution, obtaining $\delta_\phi(z') := \frac{\Pr(z'|z)}{\Pr(z')} = \frac{n_m}{n_c} \frac{q(y=1|z')}{1-q(y=1|z')}$, and modify our classifier g to accept abstract states instead of ground states. The labels are the same regardless of whether we use ground or abstract state pairs, so training will cause g to approach p and q simultaneously, thus driving $\delta_\phi(z') \to \delta(x')$ and satisfying the Density Ratio condition.

A.6 Markov State Abstractions and Kinematic Inseparability

As discussed in Section 3.1, the notion of kinematic inseparability (Misra et al., 2020) is closely related to Markov abstraction. Recall that two states x'_1 and x'_2 are defined to be kinematically inseparable if $\Pr(x, a|x'_1) = \Pr(x, a|x'_2)$ and $T(x''|a, x'_1) = T(x''|a, x'_2)$ (which the authors call "backwards" and "forwards" KI, respectively). Misra et al. (2020) define kinematic inseparability abstractions over the set of all possible "roll-in" distributions $u(x, a)$ supported on $X \times A$, and technically, the backwards KI probabilities $\Pr(x, a|x')$ depend on u. However, to support choosing a policy class, we can just as easily define u in terms of a policy: $u(x, a) := \pi(a|x)P_t^\pi(x)$. This formulation leads to:

$$P_t^\pi(x, a|x') := \frac{T(x'|a, x)\pi(a|x)P_t^\pi(x)}{\sum_{\tilde{x}\in X, \tilde{a}\in A} T(x'|\tilde{a}, \tilde{x})\pi(\tilde{a}|\tilde{x})P_t^\pi(\tilde{x})}.$$

Definition 7. *Abstraction $\phi : X \to Z$ is a* kinematic inseparability abstraction *of MDP $M = (X, A, R, T, \gamma)$ over policy class Π_C, if for all policies $\pi \in \Pi_C$, and all $a \in A; x, x'_1, x'_2, x'' \in X$ such that $\phi(x'_1) = \phi(x'_2)$; $P_t^\pi(x, a|x'_1) = P_t^\pi(x, a|x'_2)$ and $T(x''|a, x'_1) = T(x''|a, x'_2)$.*

Similarly, we can define forward—or backward—KI abstractions where only $T(x''|a, x'_1) = T(x''|a, x'_2)$—or respectively, $P_t^\pi(x, a|x'_1) = P_t^\pi(x, a|x'_2)$—is guaranteed to hold. A KI abstraction is one that is both forward KI and backward KI.

The KI conditions are slightly stronger conditions than those of Theorem 1, as the following example demonstrates.

A.6.1 Example MDP

The figure below modifies the transition dynamics of the MDP in Section 2.3.1, such that the action a_1 has the same effect everywhere: to transition to either central state, x_1 or x_2, with equal probability.

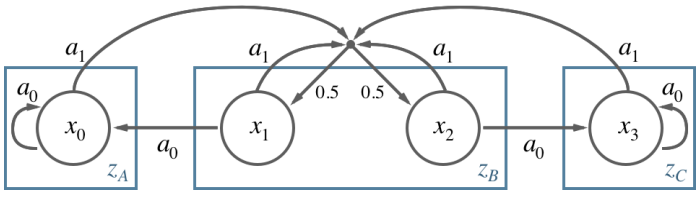


Figure A.1: An MDP and a Markov abstraction that is not a KI abstraction.

By the same reasoning as in Section 3.2.2, the Inverse Model condition holds here, but now, due to the shared transition dynamics of action a_1, the Density Ratio condition holds as well, for any policy in Π_ϕ. We can apply Theorem 1 to see that the abstraction is Markov, or we can simply observe that conditioning the belief distribution $B_{\phi,t}^\pi(x|z_B)$ on additional history has no effect, since any possible trajectory ending in z_B leads to the same 50–50 distribution over ground states x_1 and x_2. Either way, ϕ is Markov by Definition 2.

This abstraction also happens to satisfy the backwards KI condition, since $P_t^\pi(x, a|x'_1) = P_t^\pi(x, a|x'_2)$ for any (x, a) pair and any policy. However, clearly $T(x'|a_0, x_1) \neq T(x'|a_0, x_2)$, and therefore the forwards KI condition does not hold and this is not a KI abstraction.

This example shows that the Markov conditions essentially take the stance that because π is restricted to the policy class Π_ϕ, knowing the difference between x_1 and x_2 doesn't help, because π must have the same behavior for both ground states. By contrast, the KI conditions take the stance that because x_1 and x_2 have different dynamics, the agent may wish to change its behavior based on which state it sees, so it ought to choose an abstraction that does not limit decision making in that respect.

A.6.2 "Strongly Markov" Implies KI

In Section 3.3, we mentioned that the Density Ratio training objective was stronger than necessary to ensure the corresponding condition of Theorem 1. Instead of encoding the condition $\frac{\Pr(x'|z)}{\Pr(x')} = \frac{\Pr(z'|z)}{\Pr(z')}$, we discussed how the contrastive training procedure actually encodes the stronger condition $\frac{\Pr(x'|x)}{\Pr(x')} = \frac{\Pr(z'|z)}{\Pr(z')}$ that holds for each ground state x individually, rather than just in expectation. Let us call the latter condition the Strong Density Ratio condition, and call its combination with the Inverse Model condition the Strong Markov conditions.

Clearly the Strong Markov conditions imply the original Markov conditions, and, as the following theorem shows, they also imply the KI conditions.

Theorem 5. *If $\phi : X \to Z$ is an abstraction of MDP $M = (X, A, R, T, \gamma)$ such that for any policy π in the policy class Π_ϕ, both the Inverse Model condition of Theorem 1 and the Strong Density Ratio condition—i.e. $\frac{P_t^\pi(x'|x)}{P_t^\pi(x')} = \frac{P_{\phi,t}^\pi(z'|z)}{P_{\phi,t}^\pi(z')}$, for all $z, z' \in Z$; $x, x' \in X$ such that $\phi(x) = z$ and $\phi(x') = z'$—hold for every timestep t, then ϕ is a kinematic inseparability abstraction .*

Proof:

Starting from the Inverse Model condition, we have $I_t^\pi(a|x', x) = I_{\phi,t}^\pi(a|z', z)$ for all $z, z' \in Z$; $x, x' \in X$ such that $\phi(x) = z$ and $\phi(x') = z'$. Independently varying either $x' \in \phi^{-1}(z')$ or $x \in \phi^{-1}(z)$, we obtain the following:

$$[\text{Vary } x'] \to \quad I_t^\pi(a|x'_1, x) = I_t^\pi(a|x'_2, x) \tag{A.10}$$

$$[\text{Vary } x] \to \quad I_t^\pi(a|x', x_1) = I_t^\pi(a|x', x_2) \tag{A.11}$$

Similarly, if we start from the Strong Density Ratio condition, we obtain:

$$[\text{Vary } x'] \to \quad \frac{P_t^\pi(x'_1|x)}{P_t^\pi(x'_1)} = \frac{P_t^\pi(x'_2|x)}{P_t^\pi(x'_2)} \tag{A.12}$$

$$[\text{Vary } x] \to \quad \frac{P_t^\pi(x'|x_1)}{P_t^\pi(x')} = \frac{P_t^\pi(x'|x_2)}{P_t^\pi(x')} \tag{A.13}$$

If we apply Bayes' theorem to (A.12), we can cancel terms in the result, and also in (A.13), to obtain:

$$P_t^\pi(x|x'_1) = P_t^\pi(x|x'_2) \tag{A.14}$$

$$\text{and} \quad P_t^\pi(x'|x_1) = P_t^\pi(x'|x_2). \tag{A.15}$$

Combining (A.10) with (A.14), we obtain the backwards KI condition:

$$\begin{aligned} I_t^\pi(a|x'_1, x)P_t^\pi(x|x'_1) &= I_t^\pi(a|x'_2, x)P_t^\pi(x|x'_2) \\ P_t^\pi(x, a|x'_1) &= P_t^\pi(x, a|x'_2) \end{aligned} \tag{Backwards KI}$$

Similarly, we can combine (A.11) with (A.15) to obtain the forwards KI condition:

$$\begin{aligned} I_t^\pi(a|x', x_1)P_t^\pi(x'|x_1) &= I_t^\pi(a|x', x_2)P_t^\pi(x'|x_2) \\ T(x'|a, x_1)\pi(a|x_1) &= T(x'|a, x_2)\pi(a|x_2) \\ T(x'|a, x_1) &= T(x'|a, x_2) \end{aligned} \tag{Forwards KI}$$

□

Thus, we see that the training objectives in Section 3.3 encourage learning a kinematic inseparability abstraction in addition to a Markov abstraction. This helps avoid representation collapse by ensuring that we do not group together any states for which a meaningful kinematic distinction can be made.

A.7 Implementation Details for Visual Gridworld

The visual gridworld is a 6×6 grid with four discrete actions: up, down, left, and right. Observed states are generated by converting the agent's (x, y) position to a one-hot image representation (see Figure 3.1). The image displays each position in the 6×6 grid as a 3px-by-3px patch, inside of which we light up one pixel (in the center) and then smooth it using a truncated Gaussian kernel. This results in an 18×18 image (where 3px-by-3px grid cells are equidistant), to which we then add per-pixel noise from another truncated Gaussian. During pretraining, there are no rewards or terminal states. During training, for each random seed, a single state is designated to be the goal state, and the agent receives -1 reward per timestep until it reaches the goal state, at which point a new episode begins, with the agent in a random non-goal location.

A.7.1 Computing Resources

To build the figures in the chapter, we pretrained 5 different abstractions, and trained 7 different agents, each with 300 seeds. Each 3000-step pretraining run takes about 1 minute, and each training run takes about 30 seconds, on a 2016 MacBook Pro 2GHz i5 with no GPU, for a total of about 42 compute hours. We ran these jobs on a computing cluster with comparable processors or better.

A.7.2 Network Architectures

```
FeatureNet(
  (phi): Encoder(
    (0): Reshape(-1, 252)
    (1): Linear(in_features=252, out_features=32, bias=True)
    (2): Tanh()
    (3): Linear(in_features=32, out_features=2, bias=True)
    (4): Tanh()
  )
  (inv_model): InverseNet(
    (0): Linear(in_features=4, out_features=32, bias=True)
    (1): Tanh()
    (2): Linear(in_features=32, out_features=4, bias=True)
  )
  (contr_model): ContrastiveNet(
    (0): Linear(in_features=4, out_features=32, bias=True)
    (1): Tanh()
    (2): Linear(in_features=32, out_features=1, bias=True)
    (3): Sigmoid()
  )
)
QNet(
  (0): Linear(in_features=2, out_features=32, bias=True)
  (1): ReLU()
```

```
  (2): Linear(in_features=32, out_features=4, bias=True)
)
AutoEncoder / PixelPredictor(
  (phi): Encoder(
    (0): Reshape(-1, 252)
    (1): Linear(in_features=252, out_features=32, bias=True)
    (2): Tanh()
    (3): Linear(in_features=32, out_features=2, bias=True)
    (4): Tanh()
  )
  (phi_inverse): Decoder(
    (0): Linear(in_features=2, out_features=32, bias=True)
    (1): Tanh()
    (2): Linear(in_features=32, out_features=252, bias=True)
    (3): Tanh()
    (4): Reshape(-1, 21, 12)
  )
  MSELoss()
)
```

A.7.3 Hyperparameters

We tuned the DQN hyperparameters until it learned effectively with expert features (i.e. ground-truth (x, y) position), then we left the DQN hyperparameters fixed while tuning pretraining hyperparameters. For pretraining, we considered 3000 and 30,000 gradient updates (see Appendix A.10), and batch sizes within {512, 1024, 2048}. We found that the higher batch size was helpful for stabilizing the offline representations. We also did some informal experiments with latent dimensionality above 2, such as 3 or 10, which produced similar results: representations were still Markov, but harder to interpret. We use 2 dimensions in the main text for ease of visualization. We did not tune the loss coefficients, but we include ablations where either α or β is set to zero.

Hyperparameter	**Value**
Number of seeds	300
Optimizer	Adam
Learning rate	0.003
Batch size	2048
Gradient updates	3000
Latent dimensions	2
Number of conditional samples, n_c	1
Number of marginal samples, n_m	1
Loss coefficients	
$\mathcal{L}_{\text{Inverse}}$ (α)	1.0
$\mathcal{L}_{\text{Contrastive}}$ (β)	1.0
$\mathcal{L}_{\text{Smoothness}}$ (η)	0.0

Table A.2: Pretraining hyperparameters

Hyperparameter	**Value**
Number of seeds	300
Number of episodes	100
Maximum steps per episode	1000
Optimizer	Adam
Learning rate	0.003
Batch size	16
Discount factor, γ	0.9
Starting exploration probability, ϵ_0	1.0
Final exploration probability, ϵ	0.05
Epsilon decay period	2500
Replay buffer size	10000
Initialization steps	500
Target network copy period	50

Table A.3: DQN hyperparameters

A.8 Implementation Details for DeepMind Control

We use the same RAD network architecture and code implementation as Laskin et al. (2020a), which we customized to add our Markov objective. We note that there was a discrepancy between the batch size in their code implementation (128) and what was reported in the original paper (512); we chose the former for our experiments.

The SAC (expert) results used the code implementation from Yarats and Kostrikov (2020).

The DBC, DeepMDP, CPC, and SAC-AE results are from Zhang et al. (2021), except for Ball_in_Cup, which they did not include in their experimental evaluation. We ran their DBC code independently (with the same settings they used) to produce our Ball_in_Cup results.

A.8.1 Computing Resources

To build the graph in Section 3.5, we trained 4 agents on 6 domains, with 10 seeds each. Each training run (to 500,000 steps) takes between 24 and 36 hours (depending on the action repeat for that domain), on a machine with two Intel Xeon Gold 5122 vCPUs and shared access to one Nvidia 1080Ti GPU, for a total of approximately 7200 compute hours. We ran these jobs on a computing cluster with comparable hardware or better.

A.8.2 Markov Network Architecture

```
MarkovHead(
  InverseModel(
    (body): Sequential(
      (0): Linear(in_features=100, out_features=1024, bias=True)
      (1): ReLU()
      (2): Linear(in_features=1024, out_features=1024, bias=True)
      (3): ReLU()
    )
    (mean_linear): Linear(in_features=1024, out_features=action_dim, bias=True)
    (log_std_linear): Linear(in_features=1024, out_features=action_dim, bias=True)
  )
  ContrastiveModel(
    (model): Sequential(
      (0): Linear(in_features=100, out_features=1024, bias=True)
      (1): ReLU()
      (2): Linear(in_features=1024, out_features=1, bias=True)
    )
  )
  BCEWithLogitsLoss()
)
```

A.8.3 Hyperparameters

When tuning our algorithm, we left all RAD hyperparameters fixed except init_steps, which we increased to equal 10 episodes across all domains to provide adequate coverage for pretraining. We compensated for this change by adding (init_steps - 1K) catchup learning steps to ensure both methods had the same number of RL updates. This means our method is at a slight disadvantage, since RAD can begin learning from reward information after just 1K steps, but our method must wait until after the first 10 episodes of uniform random exploration. Otherwise, we only considered changes to the Markov hyperparameters (see Table A.5). We set the Markov learning rate equal to the RAD learning rate for each domain (and additionally considered 5e-5 for cheetah only). We tuned the $\mathcal{L}_{Inv}$ loss coefficient within {0.1, 1.0, 10.0, 30.0}, and the $\mathcal{L}_{Smooth}$ loss coefficient within {0, 10.0, 30.0}. The other hyperparameters, including the network architecture, we did not change from their initial values.

Hyperparameter	**Value**
Augmentation	
Walker	Crop
Others	Translate
Observation rendering	(100, 100)
Crop size	(84, 84)
Translate size	(108, 108)
Replay buffer size	100000
Initial steps	1000
Stacked frames	3
Action repeat	2; finger, walker
	8; cartpole
	4; others
Hidden units (MLP)	1024
Evaluation episodes	10
Optimizer	Adam
$(\beta_1, \beta_2) \rightarrow (\phi, \pi, Q)$	(.9, .999)
$(\beta_1, \beta_2) \rightarrow (\alpha)$	(.5, .999)
Learning rate (ϕ, π, Q)	2e-4, cheetah
	1e-3, others
Learning rate (α)	1e-4
Batch size	128
Q function EMA τ	0.01
Critic target update freq	2
Convolutional layers	4
Number of filters	32
Non-linearity	ReLU
Encoder EMA τ	0.05
Latent dimension	50
Discount γ	.99
Initial temperature	0.1

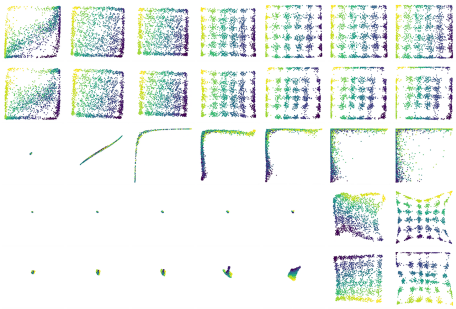
Table A.4: RAD hyperparameters

Hyperparameter	**Value**	
Pretraining steps	100K	
Pretraining batch size	512	
RAD init steps	(20K / action_repeat)	
RAD catchup steps	(init_steps - 1K)	
Other RAD parameters	unchanged	
Loss coefficients		
$\mathcal{L}_{Inv}$	30.0	ball, reacher
	1.0	others
$\mathcal{L}_{Ratio}$	1.0	
$\mathcal{L}_{Smooth}$	30.0	ball, reacher, cheetah
	10.0	others
Smoothness d_0	0.01	
Conditional samples, n_c	128	
Marginal samples, n_m	128	
Optimizer	Adam	
$(\beta_1, \beta_2) \rightarrow$ (Markov)	(.9, .999)	
Learning rate	2e-4	cheetah
	1e-3	others

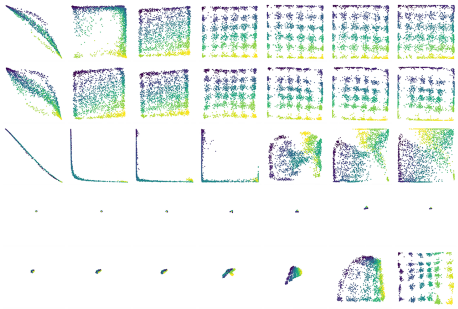
Table A.5: Markov hyperparameters

A.9 Additional Representation Visualizations

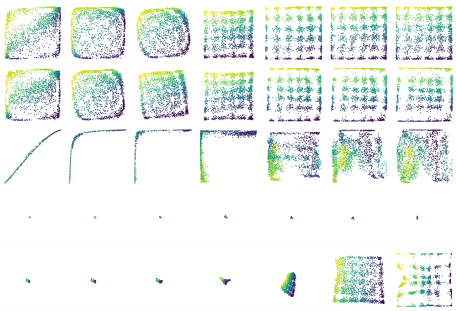
Here we visualize abstraction learning progress for the 6×6 visual gridworld domain for six random seeds. Each figure below displays selected times (after 1, 100, 200, 700, 3K, 10K, and 30K steps, progressing from left to right) of a different abstraction learning method (top to bottom): $\mathcal{L}_{Markov}$; $\mathcal{L}_{Inv}$ only; $\mathcal{L}_{Ratio}$ only; autoencoder; pixel prediction. The networks for a given random seed are initialized identically. Each point encodes a single observation. Color denotes ground-truth (x, y) position, which is not shown to the agent. Note that the third column from the right shows the representations after 3000 steps, which we use for the results in the main text. We show additional learning curves for the final representations in Appendix A.10.



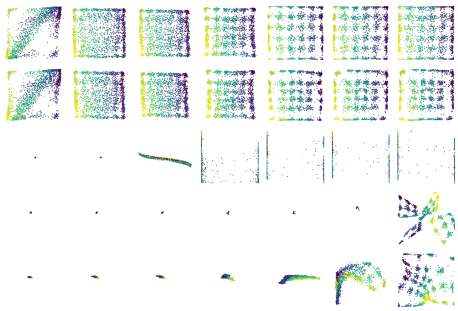
Seed 1



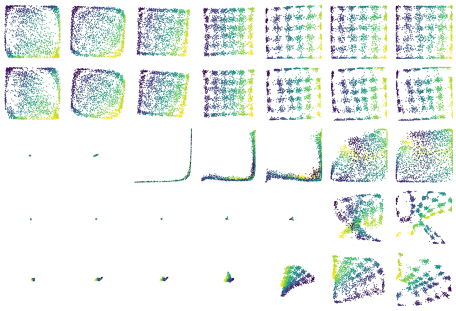
Seed 2



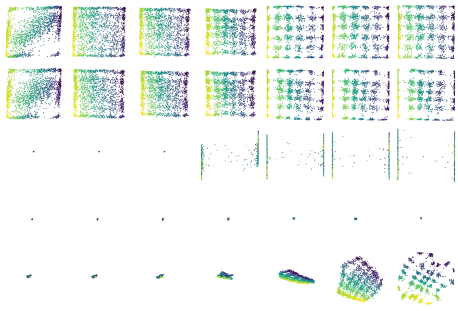
Seed 3



Seed 4



Seed 5



Seed 6

Figure A.3

A.10 Gridworld Results for Increased Pretraining Time

Since some of the representations in Appendix A.9 appeared not to have converged after just 3000 training steps, we investigated whether the subsequent learning performance would improve with more pretraining. We found that increasing the number of pretraining steps from 3000 to 30,000 improves the learning performance of ϕ_{Ratio} and $\phi_{Autoenc}$ and $\phi_{PixelPred}$ (see Figure A.4), with the latter representation now matching the performance of ϕ_{Markov}.

It is perhaps unsurprising that the pixel prediction model eventually recovers the performance of the Markov abstraction, because the pixel prediction task is a valid way to ensure Markov abstract states. However, as we discuss in Sec. 3.1.2, the pixel prediction objective is misaligned with the basic goal of state abstraction, since it must effectively throw away no information. It is clear from Figures A.3 and A.4 that our method is able to reliably learn a Markov representation about ten times faster than pixel prediction, which reflects the fact that the latter is a fundamentally more challenging objective.

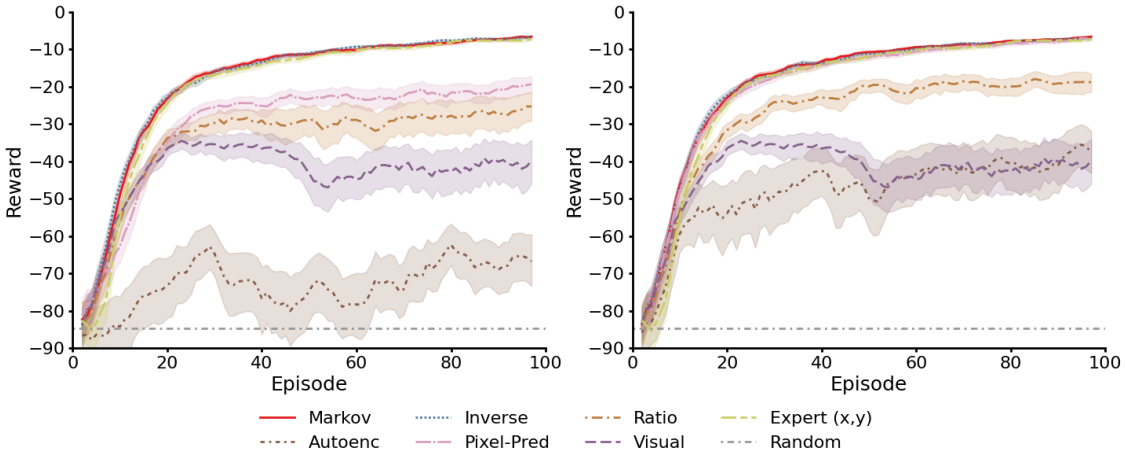


Figure A.4: Mean episode reward for the visual gridworld navigation task, using representations that were pretrained for 3,000 steps (left) versus 30,000 steps (right). Increased pretraining time improves the performance of ϕ_{Ratio}, $\phi_{Autoenc}$ and $\phi_{PixelPred}$. (300 seeds; 5-point moving average; shaded regions denote 95% confidence intervals.)

A.11 DeepMind Control Experiment with RBF-DQN

Recently, Asadi et al. (2021) showed how to use radial basis functions for value-function based RL in problems with continuous action spaces. When trained with ground-truth state information, RBF-DQN achieved state-of-the-art performance on several continuous control tasks; however, to our knowledge, the algorithm has not yet been used for image-based domains.

We trained RBF-DQN from stacked image inputs on "Finger, Spin," one of the tasks from Section 3.5, customizing the authors' PyTorch implementation to add our Markov training objective. We do not use any data augmentation or smoothness loss, and we skip the pretraining phase entirely; we simply add the Markov objective as an auxiliary task during RL. Here we again observe that adding the Markov objective improves learning performance over the visual baseline and approaches the performance of using ground-truth state information (see Figure A.5).

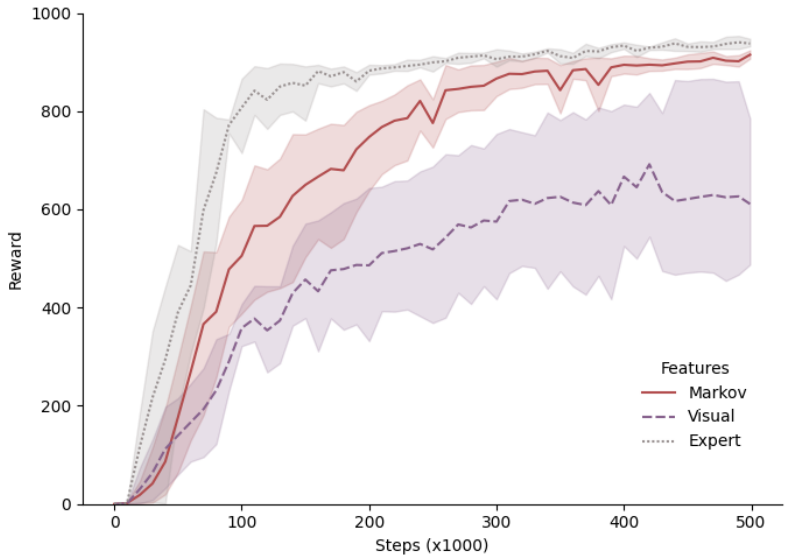


Figure A.5: Mean episode reward for RBF-DQN on "Finger, Spin" with Markov, visual, and expert features. Adding the Markov objective dramatically improves performance over the visual baseline. (Markov – 5 seeds; Visual – 6 seeds; Expert – 3 seeds; shaded regions denote 95% confidence intervals).

A.12 DeepMind Control Ablation Study

We ran an ablation study to evaluate which aspects of our training objective were most beneficial for the DeepMind Control domains. We considered the RAD implementation and its Markov variant from Section 3.5, as well as modifications to the Markov objective that removed either the pretraining phase or the smoothness loss, $\mathcal{L}_{Smooth}$ (see Figure A.6).

Overall, the ablations perform slightly worse than the original Markov objective. Both ablations still have better performance than RAD on three of six domains, but are tied or slightly worse on the others. Interestingly, removing pretraining actually results in a slight *improvement* over Markov on Finger. Removing smoothness tends to degrade performance, although, for Cheetah, it leads to the fastest initial learning phase of any method.

We suspect the results on Cheetah are worse than the RAD baseline because the experiences used to learn the representations do not cover enough of the state space. Learning then slows down as the agent starts to see more states from outside of those used to train its current representation. As we point out in Section 3.1.6, it can be helpful to incorporate a more sophisticated exploration strategy to overcome these sorts of state coverage issues. Our approach is agnostic to the choice of exploration algorithm, and we see this as an important direction for future work.

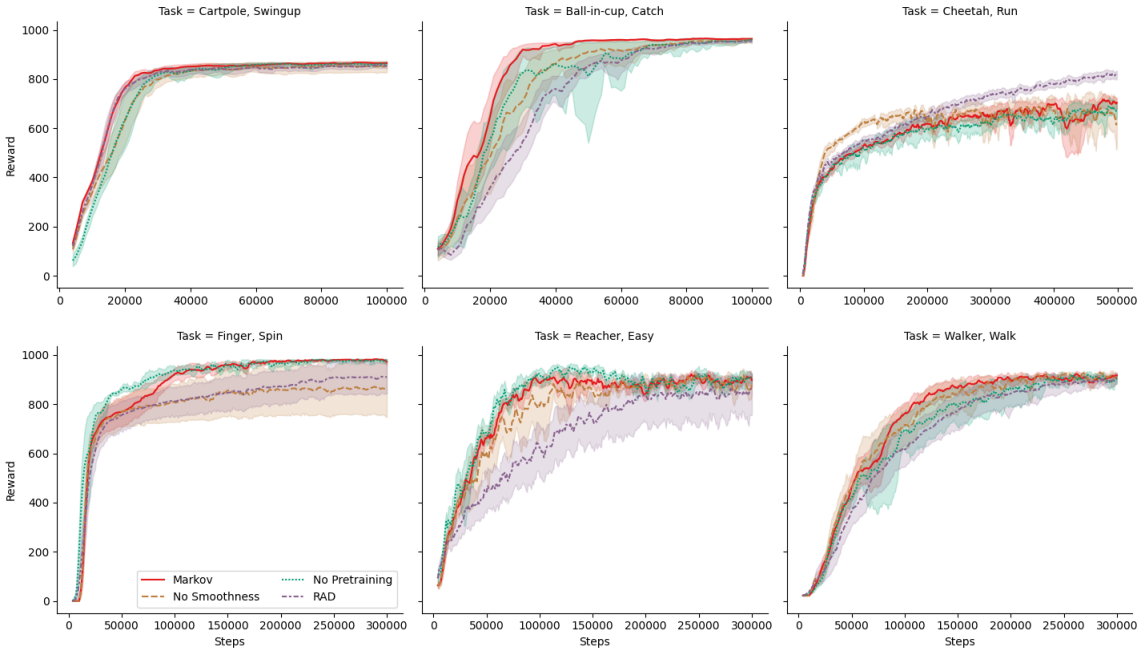


Figure A.6: Ablation results for DeepMind Control Suite. Each plot shows mean episode reward vs. environment steps. (Markov and RAD – 10 seeds; others – 6 seeds; 5-point moving average; shaded regions denote 90% confidence intervals).

Appendix B

Lambda Discrepancy

B.1 TD(λ) Fixed Point

Here we derive the fixed point of the action-value Bellman equation in a POMDP. First, define the expected n-step return given initial observation ω_0 and initial action a_0 as

$$\begin{aligned}\mathbb{E}_\Pi(G^n|\omega_0,a_0) &= \sum\nolimits_{s_0}\mathbb{P}(s_0|\omega_0)\sum\nolimits_{s_1}\mathbb{P}(s_1|s_0,a_0)\sum\nolimits_{r}\mathbb{P}(r_0|s_0,a_0,s_1)r_0+\gamma\sum\nolimits_{s_0}\mathbb{P}(s_0|\omega_0)\\ &\quad*\sum\nolimits_{s_1}\mathbb{P}(s_1|s_0,a_0)\sum\nolimits_{\omega_1}\sum\nolimits_{a_1}\sum\nolimits_{s_2}\mathbb{P}(\omega_1|s_1)\mathbb{P}(a_1|\omega_1)\mathbb{P}(s_2|s_1,a_1)\sum\nolimits_{r_1}\mathbb{P}(r_1|s_1,a_1,s_2)r_1\\ &\quad+\gamma^2\sum\nolimits_{s_0}\mathbb{P}(s_0|\omega_0)\sum\nolimits_{s_1}\mathbb{P}(s_1|s_0,a_0)\sum\nolimits_{\omega_1}\sum\nolimits_{a_1}\sum\nolimits_{s_2}\mathbb{P}(\omega_1|s_1)\mathbb{P}(a_1|\omega_1)\mathbb{P}(s_2|s_1,a_1)\\ &\quad*\sum\nolimits_{\omega_2}\sum\nolimits_{a_2}\sum\nolimits_{s_3}\mathbb{P}(\omega_2|s_2)\mathbb{P}(a_2|\omega_2)\mathbb{P}(s_3|s_2,a_2)\sum\nolimits_{r_2}\mathbb{P}(r_2|s_2,a_2,s_3)r_2+\ldots\end{aligned}$$

where we bootstrap by replacing part of the term with coefficient γ^n with a Q value. For example, with $n=2$, we replace the last line written above, or the second factor in the γ^2 term, with:

$$\sum\nolimits_{\omega_2}\mathbb{P}(\omega_2|s_2)\sum\nolimits_{a_2}\mathbb{P}(a_2|\omega_2)Q(\omega_2,a_2).$$

Translating this into matrix notation, we have

$$\begin{aligned}\left(Q_n\right)(i,a) &= \sum\nolimits_j W_{ij}\sum\nolimits_l T_{jal}R_{jal}\\ &\quad+\gamma\sum\nolimits_j W_{ij}\sum\nolimits_l T_{jal}\sum\nolimits_m\sum\nolimits_n\sum\nolimits_o\Phi_{lm}\pi_{mn}T_{lno}R_{lno}\\ &\quad+\gamma^2\sum\nolimits_j W_{ij}T_{jal}\sum\nolimits_m\sum\nolimits_n\sum\nolimits_o\Phi_{lm}\pi_{mn}T_{lno}\\ &\quad*\sum\nolimits_p\sum\nolimits_q\sum\nolimits_r\Phi_{op}\pi_{pq}T_{oqr}R_{pqr}+\ldots\end{aligned}$$

where the terms W, T, and R, are as defined in Equation 4.2, and π is the $\Omega\times A$ policy. The last line is replaced with $\sum_p\Phi_{op}\sum_q\Pi_{pq}Q_n(p,q)$ for $n=2$.

Therefore,

$$Q_n = W\left(\sum_{k=0}^{n-1}(\gamma T\Pi^S)^k R^{SA} + \gamma(\gamma T\Pi^S)^{n-1}T\Phi\Pi Q_n\right)$$

where Π is an $\Omega\times\Omega\times A$ representation of the $\Omega\times A$ policy π with $\Pi_{ijk} = \delta_{ij}\pi_{ik}$, Π^S is likewise an $S\times S\times A$ representation of the matrix $\Phi\pi$, and $R^{SA}_{ij} = \sum_l T_{ijl}R^{SAS}_{ijl}$. We also have the standard definition $Q^\lambda = (1-\lambda)\sum_{n=1}^\infty \lambda^{n-1}Q_n$, which leads to:

$$Q^\lambda = (1-\lambda)\sum_{n=1}^\infty \lambda^{n-1} W\left(\sum_{k=0}^{n-1}(\gamma T\Pi^S)^k R^{SA} + \gamma(\gamma T\Pi^S)^{n-1}T\Phi\Pi Q^\lambda\right).$$

Separating this into a reward part with factor R^{SA} and a value part with factor Q^λ, we find that the value part is

$$\begin{aligned}(1-\lambda)W\left(\sum_{n=1}^\infty(\lambda\gamma T\Pi^S)^{n-1}\right)\gamma T\Phi\Pi Q^\lambda \\ = (1-\lambda)W(I-\lambda\gamma T\Pi^S)^{-1}\gamma T\Phi\Pi Q^\lambda,\end{aligned}$$

and for the reward part, we have the coefficients of R^{SA} in the table below for values of n and k

	$k=0$	1	2	...
$n=1$	1			...
2	λ	$\lambda\gamma T\Pi^S$		...
3	λ^2	$\lambda^2\gamma T\Pi^S$	$\lambda^2(\gamma T\Pi^S)^2$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

where each term is multiplied by $(1-\lambda)W$ in front. We can then see by summing over rows before columns that the reward part is:

$$\begin{aligned}(1-\lambda)W\sum_{k=0}^\infty \frac{1}{1-\lambda}(\lambda\gamma T\Pi^S)^k R^{SA} \\ = W(I-\lambda\gamma T\Pi^S)^{-1}R^{SA}.\end{aligned}$$

So we rewrite Q^λ as follows:

$$Q^\lambda = W\left(\left(I-\lambda\gamma T\Pi^S\right)^{-1}\left(R^{SA}+(1-\lambda)\gamma T\Phi\Pi Q^\lambda\right)\right).$$

Now let $W^A = W\otimes I_{A,A}$ and $\left(W^\Pi\right)_{ijk} = \Pi W^A$. Here, $\otimes$ means the Kronecker product. This essentially repeats the W matrix A times to incorporate actions into the tensor. Note that for any $S\times A$ tensor G, $W^AG = WG$. This is because $(W^AG)_{ij} = \sum_{k,l}W^A_{ijkl}G_{kl}$, and the only nonzero terms in this sum are those such that $j=l$. For these indices, $W^A_{ijkl} = W_{ik}$, so $\sum_{k,l}W^A_{ijkl}G_{kl} = \sum_k W_{ik}G_{kj} = (WG)_{ij}$.

Also, let $F = \left(I-\lambda\gamma T\Pi^S\right)^{-1}$. Then we find:

$$\begin{aligned}Q^\lambda &= W^A\left(\left(I-\lambda\gamma T\Pi^S\right)^{-1}\left(R^{SA}+(1-\lambda)\gamma T\Phi\Pi Q^\lambda\right)\right)\\ &= W^A\left(F\left(R^{SA}+(1-\lambda)\gamma T\Phi\Pi Q^\lambda\right)\right)\\ &= W^AFR^{SA} + W^AF(1-\lambda)\gamma T\Phi\Pi Q^\lambda\end{aligned}$$

At which point we can subtract the second term on the right hand side from both sides, factor out Q^λ on the right, and multiply by $\left(I-(1-\lambda)\gamma W^A F T \Phi \Pi\right)^{-1}$ on the left of both sides to obtain:

$$\begin{aligned}
Q^\lambda &= \left(I-(1-\lambda)\gamma W^A F T \Phi \Pi\right)^{-1} W^A F R^{SA} \\
&= W^A \left(I-(1-\lambda)\gamma F T \Phi \Pi W^A\right)^{-1} F R^{SA} \\
&= W \left(I-(1-\lambda)\gamma F T \Phi W^\Pi\right)^{-1} F R^{SA} \\
&= W \left(F+(1-\lambda)\gamma F T \Phi W^\Pi F + \ldots + (1-\lambda)^k \gamma^k F T \Phi W^\Pi F T \Phi W^\Pi F + \ldots\right) R^{SA},
\end{aligned}$$

where the last equality follows from expanding the geometric series. Now we use the identity $(A-B)^{-1} = \sum_{k=0}^{\infty}(A^{-1}B)^k A^{-1}$ to find:

$$\begin{aligned}
Q^\lambda &= W \left(F^{-1}-(1-\lambda)\gamma T \Phi W^\Pi\right)^{-1} R^{SA} \\
&= W \left(I-\gamma T \left(\lambda \Pi^S + (1-\lambda)\Phi W^\Pi\right)\right)^{-1} R^{SA}.
\end{aligned}$$

To recap our previous definitions, W is an $\Omega \times S$ tensor, I is an $S \times A \times S \times A$ tensor, T is an $S \times A \times S$ tensor, Π^S is an $S \times S \times A$ tensor, Φ is an $S \times \Omega$ tensor, W^Π is an $\Omega \times S \times A$ tensor, and R^{SA} is an $S \times A$ tensor. Thus, the operation between W and the tensor inverse is a single summation over the last index of W and the first index of the inverse. Operations within the matrix inverse are likewise tensor dot products. The operation between the tensor inverse and R^{SA} is a tensor double contraction, as was the operation between W^A and tensors to its right.

Lastly, we briefly note that one can get the V values by replacing W in the above equation with W^Π and changing the operation involving it on the right from a dot product to a double contraction. We can confirm that $V_o = \sum_a \pi_{o,a} Q_{o,a}$, by rewriting the expression on the right as follows:

$$\begin{aligned}
\sum_a \pi_{o,a} Q_{o,a} &= \sum_a \pi_{o,a} \sum_{s,a'} W^A_{o,a,s,a'} B_{s,a'} \\
&= \sum_a \pi_{o,a} \sum_s W^A_{o,a,s,a} B_{s,a} \\
&= \sum_a \pi_{o,a} \sum_s W_{o,s} B_{s,a} \\
&= \sum_{s,a} \underbrace{\pi_{o,a} W_{o,s}}_{W^\Pi_{o,s,a}} B_{s,a} \\
&= V_o,
\end{aligned}$$

where $B = \left(I-\gamma T \left(\lambda \Pi^S + (1-\lambda)\Phi W^\Pi\right)\right)^{-1} R^{SA}$ is an $S \times A$ tensor.

B.2 Proof of Theorem 2 (Almost All)

In this section we prove Theorem 2, that there is either a λ-discrepancy for almost all policies or for no policies. Fix λ and λ'. Recall that we define the λ-discrepancy as follows:

$$\Delta Q_\pi := ||Q_\pi^\lambda - Q_\pi^{\lambda'}|| = \left(W^\Pi\left(A_\pi^\lambda - A_\pi^{\lambda'}\right)R^{SA}\right)\cdot w$$

where $A_\pi^\lambda = \left(I - \gamma T\left(\lambda \Pi^S + (1-\lambda)\Phi W^\Pi\right)\right)^{-1}$ and w is a weight vector of probabilities. Let U be the largest open set in the space of stochastic $\Omega \times A$ matrices, considered as a subset of $\mathbb{R}^{\Omega(A-1)}$. Now consider the lambda discrepancy as a function of the policy π. In other words, we define

$$\begin{aligned}\Delta Q : U &\to \mathbb{R} \\ \pi &\mapsto \Delta Q_\pi\end{aligned}$$

Let U be an open subset of $\mathbb{R}^n$. We say that a function $f : U \to \mathbb{R}$ is real analytic on U if for all $x \in U$, f can be written as a convergent power series in some neighborhood of x. For this proof, we will utilize the following facts: 1) the composition of analytic functions is analytic (Krantz and Parks, 2002), 2) the quotient of two analytic functions is analytic where the denominator is nonzero, 3) a real analytic function on a domain U is either identically 0 or only zero on a set of measure 0 (Mityagin, 2020).

We will also use the fact that for A an invertible matrix, each entry A^{-1}_{ij} is analytic in the entries of A where the entries of A yield a nonzero determinant. We can prove this by first writing $A^{-1} = \det(A)^{-1}\operatorname{adj}(A) = \det(A)^{-1}C^T$ where $\operatorname{adj} A$ is the adjugate of A and C is the cofactor matrix of A. Each entry of the cofactor matrix is a cofactor that is polynomial in the entries of A, and is therefore analytic in them. Therefore, each entry of A^{-1} is the quotient of two analytic functions and is therefore analytic except where $\det A = 0$.

Next, we will show that ΔQ is an analytic function. Note that the variable terms in the equation are W, W^Π, R^{SA}, T, Π^S, and Φ. R^{SA}, T, and Φ are constant with respect to π. $\Pi^S_{ilj} = \sum_k \delta_{il}\Phi_{ik}\pi_{kj}$, so each entry of Π^S is analytic on U in the entries of π. Likewise, $P_{ij} = \sum_{k,a}\Phi_{ik}\pi_{ka}T_{iaj}$ is analytic on U. Therefore, the state-occupancy counts $c = \mu_0 + \gamma P^T\mu_0 + \gamma^2(P^T)^2\mu_0 + \cdots = (I - \gamma P^T)^{-1}\mu_0$ are the composition of analytic functions and thus analytic on U. $W_{ij} = \frac{\Phi_{ji}c_j}{\sum_k \Phi_{ki}c_k}$ is analytic on U for the same reason, and the denominator of W_{ij}, $\sum_k \Phi_{ki}c_k$, is nonzero for all observations able to be observed with nonzero probability. Lastly, ΔQ is then a composition of analytic functions on U and thus analytic itself.

To finish this proof, we use the fact mentioned above that the zero set of a nontrivial analytic function is of measure zero. Therefore, the λ-discrepancy is either zero for all policies or zero only on a set of measure zero.

B.3 Proof of Lemma 4.1.1 (Block MDP)

In this section, we prove Lemma 4.1.1 concerning when the system is a block MDP. Recall that in Eq. (4.3) we define $A_\pi^\lambda = \left(I - \gamma T\left(\lambda \Pi^S + (1-\lambda)\Phi W^\Pi\right)\right)^{-1}$. Suppose $A_\pi^\lambda = A_\pi^{\lambda'}$. Then $\gamma T\left(\lambda \Pi^S + (1-\lambda)\Phi W^\Pi\right) = \gamma T\left(\lambda' \Pi^S + (1-\lambda')\Phi W^\Pi\right)$ as matrix inverses are unique. We can rewrite this as $(\lambda - \lambda')\Pi^S - (\lambda - \lambda')\Phi W^\Pi = (\lambda - \lambda')(\Pi^S - \Phi W^\Pi) = 0$. This implies that either $\lambda = \lambda'$ or $\Pi^S = \Phi W^\Pi$.

Writing $\Pi^S = \Phi W^\Pi$ out in terms of probability, this implies that for all i, j, k, $\sum_\omega \mathbb{P}(\omega|s_i)\mathbb{P}(a_k|\omega) = \sum_\omega \mathbb{P}(\omega|s_i)\mathbb{P}(a_k|\omega)\mathbb{P}(s_j|\omega)$ if $i = j$, and $\sum_\omega \mathbb{P}(\omega|s_i)\mathbb{P}(a_k|\omega)\mathbb{P}(s_j|\omega) = 0$ if $i \neq j$.

We will first consider the latter case. For all observations ω, there exists some k' such that $\mathbb{P}(a_{k'}|\omega) > 0$. We then have that for all $i \neq j$, $\sum_\omega \mathbb{P}(\omega|s_i)\mathbb{P}(a_{k'}|\omega)\mathbb{P}(s_j|\omega) = 0$. Because each term in the sum is nonnegative, this is equivalent to the statement that for all $i \neq j$ and all ω, $\mathbb{P}(\omega|s_i)\mathbb{P}(a_{k'}|\omega)\mathbb{P}(s_j|\omega) = 0$. Because $\mathbb{P}(a_{k'}|\omega)$ is positive, this implies that for all $i \neq j$ and for all ω, $\mathbb{P}(\omega|s_i)\mathbb{P}(s_j|\omega) = 0$. This means that if state s_i produces an observation o, then ω cannot be produced by any other reachable state $s_j \neq s_i$, where two states are reachable if there exists a sequence of actions sampled from the policy that enable the agent to reach state s_i from s_j with nonzero probability. In other words, we are in a block MDP for each component of the state space.

The former case doesn't add anything new. We have that for all i, k, $\sum_\omega \mathbb{P}(\omega|s_i)\mathbb{P}(a_k|\omega)(1 - \mathbb{P}(s_i|\omega)) = 0$. Because each term is nonnegative, this is equivalent to $\mathbb{P}(\omega|s_i)\mathbb{P}(a_k|\omega)(\mathbb{P}(s_i|\omega) - 1) = 0$. Because we again have that for all observations there exists an action $a_{k'}$ with nonzero probability, this means we can choose $k = k'$ to find $\mathbb{P}(\omega|s_i) = 0$ or $\mathbb{P}(s_i|\omega) = 1$ for all ω, s_i. This means that either the state s_i does not produce an observation ω, or the observation ω uniquely determines which state the agent is in.

Lastly, by going backwards through the proof, we see that the converse is also true. If the system is a block MDP, then $\Pi^S = \Phi W^\Pi$ and so $A_\pi^\lambda = A_\pi^{\lambda'}$.

B.4 Proof of Lemma 4.1.2 (Zero λ-Discrepancy)

In this section, we prove Lemma 4.1.2, deriving a condition for the λ-discrepancy to vanish.

Let $A^\lambda = \left(I - \gamma T\left(\lambda\Pi^S + (1-\lambda)\Phi W^\Pi\right)\right)^{-1}$. $Q^\lambda = Q^{\lambda'}$ iff $W(A^\lambda - A^{\lambda'})R^{SA} = 0$. Expanding A^λ and $A^{\lambda'}$ into power series, we have

$$\begin{aligned}
0 = &\left(WR^{SA} + \gamma WT\left(\lambda\Pi^S + (1-\lambda)\Phi W^\Pi\right)R^{SA} + \right.\\
&\quad \left.\gamma^2 WT\left(\lambda\Pi^S + (1-\lambda)\Phi W^\Pi\right)T\left(\lambda\Pi^S + (1-\lambda)\Phi W^\Pi\right)R^{SA} + \ldots\right)\\
&- \left(WR^{SA} + \gamma WT\left(\lambda'\Pi^S + (1-\lambda')\Phi W^\Pi\right)R^{SA} + \right.\\
&\quad \left.\gamma^2 WT\left(\lambda'\Pi^S + (1-\lambda')\Phi W^\Pi\right)T\left(\lambda'\Pi^S + (1-\lambda')\Phi W^\Pi\right)R^{SA} + \ldots\right)
\end{aligned}$$

We observe that it is potentially possible to get cases of zero λ-discrepancy from the terms in this equation cancelling out. This can occur in one particularly nice way if $\lambda = 0$ and $\lambda' = 1$. In this case, there is no λ-discrepancy precisely when

$$\begin{aligned}
0 &= \left(WR^{SA} + \gamma WT\Pi^S R^{SA} + \gamma^2 WT\Pi^S T\Pi^S R^{SA} + \ldots\right)\\
&\quad - \left(WR^{SA} + \gamma WT\Phi W^\Pi R^{SA} + \gamma^2 WT\Phi W^\Pi T\Phi W^\Pi R^{SA} + \ldots\right)\\
&= \gamma\left(WT\Pi^S R^{SA} - WT\Phi W^\Pi R^{SA}\right)\\
&\quad + \gamma^2\left(WT\Pi^S T\Pi^S R^{SA} - WT\Phi W^\Pi T\Phi W^\Pi R^{SA}\right) + \ldots
\end{aligned}$$

This occurs when each pair of n-step return terms cancel, or

$$\begin{aligned}
WT\Pi^S R^{SA} - WT\Phi W^\Pi R^{SA} &= 0\\
WT\Pi^S T\Pi^S R^{SA} - WT\Phi W^\Pi T\Phi W^\Pi R^{SA} &= 0\\
&\ldots
\end{aligned}$$

Interpreting this in terms of probabilities, this says that for all initial observations ω^0, initial actions a^0, and horizons T,

$$\mathbb{E}(r^T|\omega^0, a^0) = \sum\nolimits_{\omega^1,\ldots,\omega^T} \mathbb{E}(r^T|\omega^T)\mathbb{P}(\omega^1|\omega^0, a^0)\prod_{t=1}^{T-1}\mathbb{P}(\omega^{t+1}|\omega^t) \tag{B.1}$$

In particular, this applies when R^{SA} is a constant matrix, as in this case, the expected rewards are all equal. Therefore, when R^{SA} is constant, $Q^1 = Q^0$.

B.5 Lambda Discrepancy Norm Weighting

The λ-discrepancy introduced in Definition 3 and formalized in Equation 4.5 contains a weighted norm over the observations and actions of the decision process. There are many choices of norm and weighting scheme. We use an L^2 norm to highlight the connection to mean-squared action-value error. For the weighting scheme, we weight actions according to the policy for each observation, and we weight observations uniformly. We also considered weighting observations according to their discounted visitation frequency, but found that this led to worse performance during memory optimization.

B.6 Analytical Memory Optimization

B.6.1 Memory-Augmented POMDP

As referenced in Section 4.2.1, here we will explain how to define a memory-augmented POMDP from a base POMDP $(S, A, T, R, \Omega, \Phi, \gamma)$. Given a set of memory states M, we will augment the POMDP as follows:

$$S_M = S \times M$$

$$A_M = A \times M$$

$$\Omega_M = \Omega \times M$$

$$T_M : S_M \times A_M \times S_M \to [0,1], (s_0, m_0) \times (a_0, m_1) \times (s_1, m_2) \mapsto T(s_0, a_0, s_1)\delta_{m_1 m_2}$$

$$R_M : S_M \times A_M \to [0,1], (s_0, m_0) \times (a_0, m_1) \mapsto R(s_0, a_0)$$

$$\Phi_M : S_M \times \Omega_M \to [0,1], (s_0, m_1) \times (\omega_0, m_2) \mapsto \Phi(s_0, \omega_0)\delta_{m_1 m_2}$$

$$\gamma_M = \gamma$$

This augmentation scheme uses the memory states M in three ways: as augmentations of states, actions, and observations. The state augmentation concatenates the environment state S with the agent's internal memory state M. Meanwhile, the action augmentation A_M provides the agent with a means of managing its internal memory state. Together, these augmentations allow writing the augmented transition dynamics T_M, which are defined so as to preserve the underlying state-transition dynamics T while allowing the agent full control to select its desired next memory state. The observation augmentation Ω_M provides the agent with additional context with which to make policy decisions, and the observation function Φ_M preserves the original behavior of the observation function Φ while giving the agent complete information about the internal memory state.

We define an augmented policy π as follows:

$$\pi : \Omega_M \times A_M \to [0,1],$$

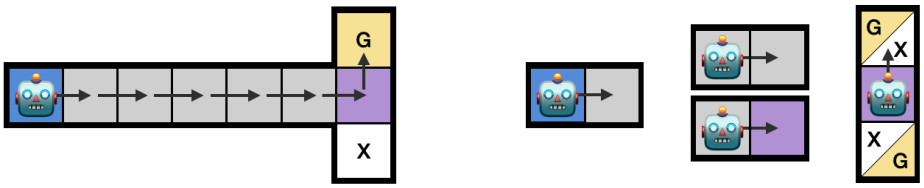


Figure B.1: Visualizations of value functions computed using MC (left) and TD (right). MC averages over entire trajectories, so it can associate the blue observation with the upward goal. By contrast, TD computes value by bootstrapping; its value estimates for subsequent observations ignore any prior history.

which we decompose into two parts, an action policy and a memory policy:

$$\pi(a, m'|\omega, m) = \pi_A(a|\omega, m)\pi_M(m'|a, \omega, m),$$

$$\pi_A : \Omega \times M \times A \to [0, 1],$$

$$\pi_M : \Omega \times M \times A \times M \to [0, 1].$$

Note that the memory policy π_M has the same function signature as—and is equivalent to—the state-machine formulation of memory functions introduced in Section 2.3.2. This definition of memory functions is convenient because it allows us to express a memory-augmented POMDP as a new POMDP using simple Cartesian products.

Now we can define the value function over the Cartesian product of observations and memories:

$$Q^{\lambda}_{\pi_M} = W_M \Big(I - \gamma T_M \big(\lambda \Pi^S + (1 - \lambda) \Phi W^{\Pi}_M \big) \Big)^{-1} R_M, \tag{B.2}$$

where all quantities (including W_M and W^{Π}_M) have been modified to handle the augmented states and observations. We provide pseudocode for taking this memory-Cartesian product of a POMDP in Appendix B.6.3, Algorithm 5.

B.6.2 Environments and Analytical Algorithms

B.6.2.1 T-Maze Details

We use T-maze with corridor length 5 as an instructive example. The environment has 15 underlying MDP states: one initial state for each reward configuration (reward is either up or down), five for each corridor, one for each junction, and finally the terminal state. There are 5 observations in this environment - one for each of the initial states, a corridor observation shared by all corridor states, a junction observation shared by both junction states, and a terminal observation. The action space is defined by movement in the cardinal directions. If the agent tries to move into a wall, it remains in the current state. From the junction state, the agent receives a reward of $+4$ for going north, and -0.1 for going south in the first reward configuration. The rewards are flipped for the second configuration. The environment has a discount rate of $\gamma = 0.9$.

This environment makes it easy to see the differences between MC and TD approaches to value function estimation. We visualize these differences in Figure B.1. MC computes the average value for each observation by averaging the return over all trajectories starting from that observation. By contrast, TD averages over the 1-step observation transition dynamics and rewards, and bootstraps off the value of the next observation. For a policy that always goes directly down the corridor and north at the junction, this leads to an average return for the blue observation of $+4$ with MC and $(4 - 0.1)/2 = 1.95$ with TD (ignoring discounting).

B.6.2.2 Other POMDP Details

All other POMDPs used in the experiments Section 4.2.1.2 were taken from pre-defined POMDP definitions (Cassandra, 2003). The only exception is the Tiger environment, where we preserve the underlying environment behavior, but adapt the domain specification to match our formalism such that observations are only a function of state.

The original Tiger domain used a hand-coded initial belief distribution that was uniform over the two states L/R, and did not emit an observation until after the first action was selected. Thereafter, the observation function was action-dependent, with state-action pair $(\texttt{L}, \texttt{listen})$ emitting observations `left` and `right` with probability 0.85 and 0.15 respectively, and other actions $(\texttt{L}, \texttt{*})$ emitting uniform observations and returning to the initial belief distribution. Since our agent does not have access to the set of states, it cannot use an initial belief distribution. To achieve the same behavior, we modified the domain by splitting each state L/R into an initial state $\texttt{L}_1$/$\texttt{R}_1$ that always emits an `initial` observation, and a post-listening state $\texttt{L}_2$/$\texttt{R}_2$ that uses the 0.85/0.15 probabilities. We visualize these changes in Figure B.2. This type of modification is always possible for finite POMDPs and does not change the underlying dynamics.

B.6.2.3 Memory Improvement Algorithm

In this section we provide pseudocode for the memory-learning algorithm described in Section 4.2.1.1 in Algorithm 4. This function takes as input a POMDP $\mathcal{P}$ and memory parameters θ_μ, and minimizes the observation-space λ-discrepancy as defined in Equation 4.6. This minimization is achieved through a gradient descent update computed using the auto-differentiation package JAX (Bradbury et al., 2018).

Algorithm 4 Memory Improvement

Input: Fixed policy parameters θ_π, where $\Pi = \mathrm{softmax}(\theta_\pi)$, memory parameters θ_μ, POMDP parameters $\mathcal{P} := (T, R^{SA}, \phi, p_0, \gamma)$, number of improvement steps $n_{\text{steps},\mu}$, learning rate $\alpha \in [0, 1]$
for $i = 0$ **to** $n_{\text{steps},M} - 1$ **do**
// Augment MDP with memory parameters θ_μ
$\mathcal{P}_{\theta_\mu} \leftarrow$ `expand_over_memory`$(\mathcal{P}, \theta_\mu)$
// Calculate MC (no memory augmentation) and TD (with memory augmentation) value functions.
$Q^1_\pi = W\left(I - \gamma T \Pi^S\right)^{-1} R^{SA},\ Q^0_{\pi_\mu} = W_M\left(I - \gamma T_M \Phi_M W_M^\Pi\right)^{-1} R_M$
// Map $\Omega \times M$-space value function back to Ω-space with $p(m \mid o), \forall o \in \Omega$
$\hat{Q}^0_\pi = \sum_{m \in M} p(m|\cdot) Q^0_{\pi_\mu}$
// Calculate the λ-discrepancy
$Q^\lambda_{\pi_\mu} = ||\hat{Q}^0_{\pi_\mu} - Q^1_\pi||_{\pi_\theta, 2}$
// Calculate the gradient of $Q^\lambda_{\pi_\mu}$ w.r.t. θ_M, update memory parameters
$\theta_\mu \leftarrow$ `update_params`$(\alpha, \theta_\mu, \nabla_{\theta_\mu} Q^\lambda_{\pi_\mu})$
end for
return θ_M

Here, `update_params()` is any gradient-descent-like update, such as stochastic gradient descent or Adam etc. As a note, all parameters $\boldsymbol{\theta}$ in these experiments are initialized with a Gaussian distribution, with mean 0 and standard deviation 0.5.

B.6.3 Analytical Memory Cartesian Product

In this section, we define the memory-Cartesian product function, `expand_over_memory()`, used by Algorithm 4. This function computes the Cartesian product of the POMDP $\mathcal{P}$ and the memory state space M, as described in Appendix B.6.1.

Algorithm 5 Memory Cartesian Product (`expand_over_memory`)

Input: Memory parameters θ_M (with corresponding memory function M), POMDP parameters $\mathcal{P} := (T, R^{SA}, O, p_0, \gamma)$, number of memory states $|M|$
// Repeat reward function for each state over each memory $m \in M$.
$R^{SA}_M \leftarrow$ `repeat_over_states`$(R^{SA}, |M|)$
// Calculate transition function cross product.
$T^O_M \leftarrow$ `einsum`$('ij, jklm \to iklm', O, M)$
$T_M \leftarrow$ `einsum`$('iljk, lim \to lijmk')$
// Calculate observation function cross product. $I_{|M|}$ is the identity function over $|M|$.
$O_M \leftarrow$ `kron`$(O, I_{|M|})$
// Finally, calculate the initial state distribution.
$p_{0,M=0} \leftarrow p_0$
return $(T_M, R^{SA}_M, O_M, p_{0,M}, \gamma)$

Note that `einsum` is the Einstein summation, and `kron` is the Kronecker product.

B.6.4 Memory Optimization for Value Improvement

Now we describe the overall optimization process, incorporating both memory improvement and policy improvement. We start with policy iteration to learn a TD-optimal memoryless policy. Then we augment the policy and POMDP to incorporate the memory state space M. Next, we improve memory by minimizing the λ-discrepancy. Finally, we run policy iteration again on the memory-augmented POMDP.

Algorithm 6 Memory Optimization with Value Improvement

Input: Randomly initialized policy parameters θ_π, where $\Pi = \mathrm{softmax}(\theta_\pi)$, randomly initialized memory parameters θ_μ, POMDP parameters $\mathcal{P} := (T, R^{SA}, \phi, p_0, \gamma)$, number of memory improvement steps $n_{\text{steps},M}$, number of policy iteration steps $n_{\text{steps},\pi}$, learning rate $\alpha \in [0, 1]$
// Calculate memoryless optimal policy.
$\theta_\pi \leftarrow$ `policy_iteration`$(\theta_\pi, \mathcal{P}, n_{\text{steps},\pi})$
// Repeat policy over all memory states.
$\theta_{\pi_\mu} \leftarrow$ `repeat`$(\theta_\pi, |M|)$
// Improve memory function.
$\theta_\mu \leftarrow$ `memory_improvement`$(\theta_\mu, \theta_{\pi_\mu}, \mathcal{P}, n_{\text{steps},M})$
// Improve memory-augmented policy over learnt-memory-augmented POMDP
$\theta_{\pi_\mu} \leftarrow$ `policy_iteration`$(\theta_{\pi_\mu}, \mathcal{P}_{\theta_\mu}, n_{\text{steps},\pi})$
return $\theta_{\pi_\mu}, \theta_\mu$

In this algorithm, `policy_iteration()` runs for a fixed number ($n_{\text{steps},\pi}$) of steps, alternating between analytical policy evaluation and updating the policy to be ϵ-greedy with respect to the resulting Q-function.

B.6.5 Experiment Details

B.6.5.1 Analytical Memory Optimization Experiment Details

For all experiments in Section 4.2.1.2, we run memory optimization on the suite of POMDPs with the following hyperparameters. We optimize memory for $n_{\text{steps},M} = 20K$ steps and run policy iteration for $n_{\text{steps},\pi} = 10K$ steps. For all analytical experiments, we use the Adam optimizer (Kingma and Ba, 2015).

For the belief-state baselines, solutions were calculated using a POMDP solver from the `pomdp-solve` package (Cassandra, 2003). The belief-state solution for the 4×3 maze was solved using an epsilon parameter of $\epsilon = 0.01$, due to convergence issues with the environment when utilizing POMDP solvers. We were unable to compute the belief-state solution for Hallway, as belief-state algorithms have historically performed extremely poorly on that domain; consequently, the upper normalization performance in Figure 4.2b uses the value of an optimal ground-truth-state policy, learnt using value iteration, rather than an optimal belief-state policy.

B.6.5.2 Discussion of Tiger Results

We now briefly discuss the performance of Tiger in Figure 4.2b—the only environment where our algorithm does not improve performance. In Tiger, the optimal memoryless policy is to `listen` initially, followed by opening whichever door the tiger is not in. This policy has the same value function under both TD and MC, and so *has no λ-discrepancy*. Because our approach first computes the TD-optimal memoryless policy, then uses the resulting λ-discrepancy to optimize memory, the lack of λ-discrepancy means our memory optimizer has no gradient signal to follow. As a result, we see the same performance across all numbers of memory states. This failure mode stems from the interaction between the way we select the policy for memory optimization and the inherent symmetry of the Tiger problem. Theorem 2 implies that such failures will be rare, and that they can be avoided by choosing a different policy.

To show that the Tiger failure mode is an artifact of this specific environment and not of this form of partial observability, we compare the performance of our analytical method on the Tiger problem versus the Tiger Grid problem (Littman et al., 1995) in Figure B.3. Tiger Grid is a scaled-up, grid-based version of Tiger with the same underlying problem of resolving uncertainty about which door contains the tiger. Future work should investigate more robust ways to select the policy used for memory optimization.

B.7 Sample-Based Experiments

B.7.1 Sample-Based Results on Tabular POMDPs

In the experiments of Figure 4.3, we train all agents using the online SARSA algorithm (Rummery and Niranjan, 1994). For all environments, we terminate episodes with probability $(1-\gamma)$ at each timestep, which is equivalent to reward discounting (Puterman, 1994). We adjust the weight of the lambda discrepancy in our optimization objective $\mathcal{L}$ using the hyperparameter $\eta \geq 0$:

$$\mathcal{L} = (v_1 - v_0)^2 + 2(v_1 - \hat{v}_0)(v_0 - \hat{v}_1) + (1+\eta)(\hat{v}_1 - \hat{v}_0)^2, \tag{B.3}$$

where v_0, v_1, $\hat{v}_0$ and $\hat{v}_1$ are defined in Section 4.2.2. The two curves representing our method both use this objective function to train, however one chooses actions using the Q-values from the TD head, and one using those from the MC head. For a given episode, the agent chooses the maximum-valued (TD/MC) action with probability $(1-\epsilon)$ and a random action with probability ϵ. At the end of each episode, each method updates its Q-function by following a single step of an optimizer, using the data gathered during that episode and the objective function defined by the method.

We choose a different value of η for the two versions of our method, and use the chosen value for all environments. After sweeping over $\eta \in \{0.0, \text{1e-5}, \text{1e-4}, \text{1e-3}, \text{1e-2}, \text{1e-1}, \text{1e0}, \text{1e1}\}$, we choose $\eta = 0.1$ for the MC-action-selection head, and $\eta = 1.0$ for the TD-action-selection head. As in prior work training LSTMs

on T-Maze (Bakker, 2001), all experiments use an LSTM layer of dimension 12. The hyperparameters listed below are shared by all four experiments and were chosen without tuning on any method.

Parameter Name	Parameter Value
LSTM Layer Size	12
Learning Rate	0.001
Optimizer	Adam
γ	0.95
ϵ	0.1

Table B.1: Shared LSTM Hyperparameters

The natural reward scales of POMDPs in this suite can be quite large, which presents difficulty for neural-network methods. We employ a simple normalization scheme, where the agent normalizes the reward for an environment by the range that rewards may span: $r_{\text{agent}} = r/(r_{\max} - r_{\min})$. This preserves the sign of the original rewards while ensuring that the normalized rewards fall within the range $[-1, 1]$.

B.7.2 Sample-Based Results on RockSample

To assess the scalability of our approach, we consider the RockSample(7, 8) environment (Smith and Simmons, 2004)—an environment with non-tabular features and a state space orders of magnitude larger than the other POMDPs we consider. We plot learning curve results in Figure B.4, and see that both versions of our method outperform the agents that use either MC or TD value functions alone. This result suggests that minimizing λ-discrepancy is a promising approach for resolving partial observability in large domains.

B.7.3 Experimental details for RockSample

In our experiments for RockSample, we limit the maximum episode length to 1000 environment time steps. The rock locations are also randomly sampled and fixed for each seed. The remaining settings and hyperparameters are the same as the original formulation of RockSample(7, 8) (Smith and Simmons, 2004).

As in the other LSTM experiments, (see Appendix B.7.1), we use the SARSA algorithm, but here we add an experience replay buffer (Lin, 1992). The LSTM hyperparameters were selected based on best performance for the TD agent, and for the λ-discrepancy agents, we simply selected $\eta = 1$ without any tuning. The feature space uses action concatenation, where the one-hot representation of the previous action is concatenated to the LSTM input. Since we are using a replay buffer, we also use truncated-backpropagation through time (T-BPTT) (Werbos, 1990) for only this domain.

Parameter Name	Parameter Value
LSTM Layer Size	100
Learning Rate	0.0001
Optimizer	Adam
γ	0.99
ϵ	0.1
T-BPTT Truncation	10
Buffer Size	100K

Table B.2: Shared LSTM Hyperparameters for RockSample

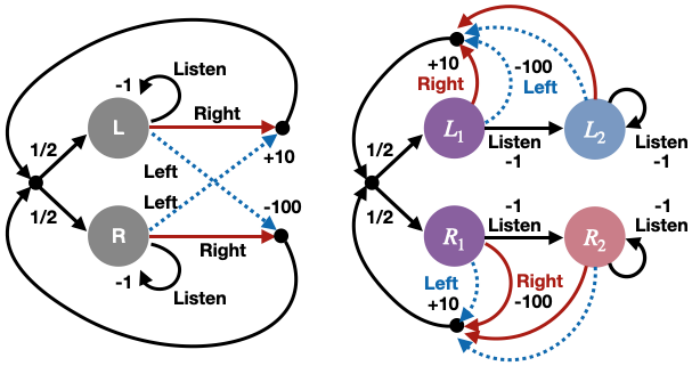


Figure B.2: Visualizations of the Tiger POMDP. In the original version (left) the observation function was action-dependent, whereas in our modified version (right) observations only depend on state. The state color for the domain on the right represents the distinct state-dependent observation functions: purple states use the `initial` observation, while the other states are biased towards either `left` (blue) or `right` (red) observations with probability 0.85.

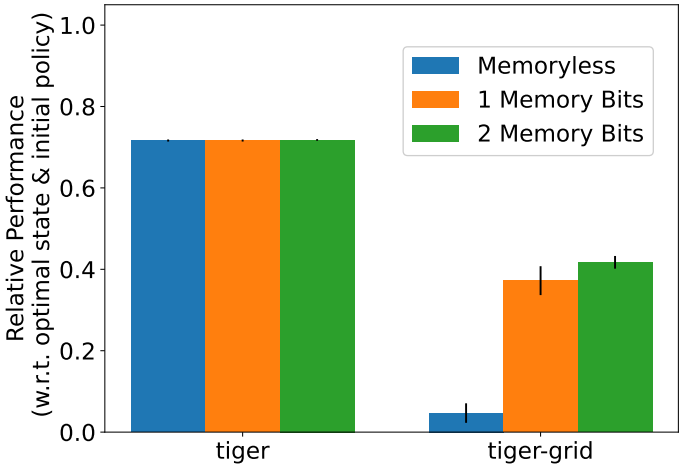


Figure B.3: Results on Tiger and Tiger Grid.

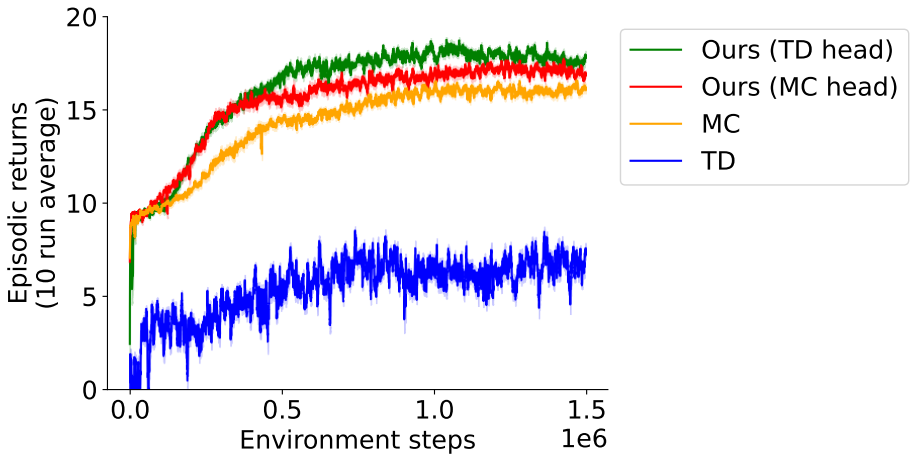


Figure B.4: RockSample results. (Individual learning curves are averaged over a 5000-step window.)

Appendix C

Focused Macro-Actions

C.1 Suitcase Lock Implementation

Each Suitcase Lock problem instance has N dials, each with M digits, and $2N$ actions, half which increment a deterministic subset of the dials (modulo M), and half which decrement the same dials (see Figure 5.2). Let k_i denote the effect size of action a_i, and $\bar{k}$ denote the mean effect size across all actions. Given a $\bar{k}$, we generate problem instances with different start states, goal states, and sets of actions such that they have mean effect size $\bar{k}$.

We always ensure that every state can be reached from every other state. Note that if $\bar{k} = N$, or if all actions modify (for example) an even number of state variables, it is not possible to reach every state from every other state. To circumvent this issue, we check that for a given problem instance, the increment and decrement action sets can each be reduced to an $N \times N$ binary matrix with full rank. We repeatedly generate action sets with the desired mean effect size until we find one that satisfies this condition. The resulting action sets are therefore different for each random seed, except when $\bar{k} = 1$ where we always use the identity matrix I, and when $\bar{k} = (N - 1)$ where we use $1 - I$ with an extra 1 added to the first diagonal element to break symmetry. The decrement actions are always the negation of the increment actions, and we ignore them for $M = 2$.

C.2 Simulator Details

C.2.1 PDDLGym

We use the PDDLGym library Silver and Chitnis, 2020 to automatically construct black-box simulators for PDDL planning problems. State information is represented as a variable-length list of currently-true literals. The planning agent has access to this state information, along with the goal (represented as a conjunction of literals), the action applicability function, and the simulator function. We chose a representative set of PDDL problems and executed uniform random actions to generate 100 unique random starting states for each, keeping the goal fixed. The associated `.pddl` files can be found in the code repository.

C.2.2 15-Puzzle

The 15-puzzle is a 4×4 grid of 15 numbered, sliding tiles and one blank space (see Figure C.1a). The puzzle begins in a scrambled configuration, and the objective is to slide the tiles until the numbers are arranged in increasing order. There are approximately 10^{13} states and the worst-case shortest solution requires 80 actions Brüngger et al., 1999. Our simulator uses a state representation with 16 variables (for the positions of each tile and of the blank space), and 48 primitive actions (that swap the blank space with one of the adjacent tiles), of which only 2–4 can be applied in each state. Similarly, macro-actions can only run if they begin with the correct blank space location.

We set the macro-learning budget $B_M = 32{,}000$ simulator queries, the number of macros $N_M = 192$, and the number of repetitions $R_M = 16$. The budget was chosen to approximately match the number of steps required to solve one problem instance with primitive actions. This resulted in 12 generated macros per repetition, and a per-repetition simulator budget of 2000 state transitions. We compared these macro-actions against 192 "random" macro-actions of the same lengths, which were generated (for each random seed) by selecting actions uniformly at random from the valid actions at each state.

We then solved the 15-puzzle using greedy best-first search with the goal-count heuristic and a simulation budget of $B_S = 500{,}000$ state transitions. We generate 100 unique starting states by scrambling the 15-puzzle with uniform random actions for either 225 or 226 steps, with equal probability (to ensure that we see all possible blank space locations). The resulting puzzles can be found in the code repository.

Note C.2.2.1. ***On Finding States Where Macro Preconditions Do Not Apply***

As mentioned in Section 5.2, the macro-learning procedure includes an option to repeat the search R_M times from new starting states where the previously-discovered macros are not applicable. In general, finding such a state can be as hard as planning, although it might be easier if there is no requirement for generating a plan, e.g. by resetting the simulator to generate a new starting state. Some environment implementations do not allow resetting to arbitrary states, and, in those cases, a plan must be generated. 15-puzzle is the only domain

where we use $R_M > 1$, and for this domain, we found that either state generation strategy was effective. For domains where the simulator cannot be reset, and where a random walk is insufficient, it is possible to make the search more informed, such as by incorporating state novelty into the heuristic.

C.2.3 Rubik's Cube

The Rubik's cube is a $3 \times 3 \times 3$ cube with colored stickers on each outward-facing square (see Figure C.1b). The puzzle begins in a scrambled configuration, and the objective is to rotate the faces of the cube until all stickers on each face are the same color. There are approximately 4.3×10^{19} states, and the worst-case shortest solution requires 26 actions Rokicki, 2014. Our simulator fixes a canonical orientation of the cube, and uses a 48-state-variable representation (for the positions of each colored square, excluding the stationary center squares). The problem has 12 primitive actions (i.e. rotating each of the 6 faces by a quarter-turn in either direction), and these actions are highly non-focused: each modifies 20 of the 48 state variables.

We set the number of learned macro-actions $N_M = 576$ so that we could fairly compare the generated macro-actions against our set of expert macro-actions. We learned macro-actions from a single starting state $R_M = 1$, and set a simulation budget of $B_M = 1{,}000{,}000$ simulator queries. We also compared against 576 "random" macro-actions of the same lengths as the expert macros (six distinct macro-actions plus their corresponding variations), which were regenerated for each random seed. We set the search budget $B_S = 2{,}000{,}000$ simulator queries.

We obtained starting states for Rubik's cube from modified versions of the 100 hardest problems from Büchner (2018). The problems were specified as random sequences of primitive actions to be applied to a solved Rubik's cube in order to generate the starting state, as well as a corresponding SAS$^+$ representation for each problem. The original Büchner problems incorporated 18 half-turn and quarter-turn action primitives, whereas our simulator uses only 12 quarter-turn action primitives. Our modification removed the 6 half-turn actions from the SAS$^+$ representation and converted problem specifications involving half-turns to their equivalent quarter-turn-only specifications. The resulting problems consisted of between 12 and 29 primitive actions, with an average of about 20. (We also tried generating starting states by scrambling the cube with uniform random actions for 60 steps, with similar results.) The problems we use, and the procedure we use to generate randomly scrambled starting states, can be found in the linked code repository.

(a) 15-Puzzle

(b) Rubik's Cube

Figure C.1: Visualizations of the planning domains that use domain-specific simulators

C.3 Updating the Simulator with Macros

PDDLGym

For the PDDLGym simulators, we build new macro-operators for the saved primitive-action sequences by:

1. Re-binding the original lifted parameters to new variables that capture any dependencies between subsequent actions. For example, the sequence [`PLACE_ON(B,C)`, `PLACE_ON(A,B)`], would result in two distinct parameters for objects `A` and `C`, plus a third, shared parameter for object `B` that is reused by both primitives.

2. Combining the preconditions of subsequent primitive actions when they are not already met by the effects of previous primitive actions. For example, if `ACTION1` has precondition `(A and B)` and effect `C`, and `ACTION2` has precondition `(C and D)`, this would result in the combined precondition `(A and B and D)`.

3. Combining and simplifying the effects of the primitive actions to remove unnecessary negations. For example, if the combined precondition so far is `A`, and if `ACTION1` has effect `(B and (not A))` and `ACTION2` has effect `(C and A)`, this would result in a combined effect of `(B and C)`, since `A` is already a precondition.

We present pseudocode in Algorithm 2, and the implementation and the resulting macro-augmented PDDL files can be found in the code repository.

Note that while the desired number of macro-actions for all PDDLGym domains was set to $N_M = 8$, we were only able to find four unique macros for the `doors` domain.

Domain-Specific Simulators

For 15-puzzle and Rubik's cube, both simulators use a position-based representation (i.e. the positions of each numbered tile or blank space; the positions of each colored sticker excluding the stationary center stickers). Primitive actions are expressed as permutations operations on the indices of the state variables.

To augment the simulator with macro-actions, we computed the overall permutation for each sequence of primitive actions, and store the result (along with its precondition, if any) as a new permutation operation that the simulator can apply using the same procedure it uses for primitive actions.

In the case of Rubik's cube, none of the primitive actions have preconditions, so the resulting macros do not have preconditions either. However, for 15-puzzle, primitive-action preconditions depend on the position of the blank space. Fortunately, since we only construct macros for valid action sequences and since actions deterministically modify the position of the blank space, as long as the initial precondition is satisfied, each action will automatically satisfy the precondition of the next action in the sequence. Thus, when saving each

Algorithm 2 Construct lifted macro for PDDLGym

Input:
actions, a sequence of grounded primitive actions
operators, map from names to lifted primitive operators
Output:
macro, a newly-constructed, lifted macro-operator

```
1:  macro.params := ∅
2:  macro.preconds := ∅
3:  macro.effects := ∅
4:  lifted := map from grounded to lifted variable names
5:  for action in actions do
6:      op := operators[action.name]
7:      lifted.update({ v ↦ new_variable_name(), for v in action.variables if v not in lifted})
8:      binding := {p ↦ lifted[v], for (p, v) in zip(op.params, action.variables)}
9:      for p in op.params do
10:         macro.params.add( binding[p] )
11:     end for
12:     for literal in bind_literals(op.preconds, binding) do
13:         if literal not in macro.effects and literal not in macro.preconds then
14:             macro.preconds.add(literal)
15:         end if
16:     end for
17:     cleanup_contradictory_effects(op.effects)
18: // Simplify any contradictory effects to just their positive part, e.g. ((not A) and A) becomes (A)
19:     for literal in bind_literals(op.effects, binding) do
20:         if (¬literal) in macro.effects then
21:             macro.effects.remove(¬literal)
22:         else
23:             macro.effects.add(literal)
24:         end if
25:     end for
26: end for
27: return macro
```

15-puzzle macro-action, we simply keep track of the blank-space location required to execute its first primitive action, along with its overall permutation.

The code to generate the overall permutation of a 15-puzzle or Rubik's cube macro-action can be found in the corresponding module in the code repository.

C.4 Expert Rubik's Cube Macros

We use the following expert macro-actions (expressed in standard cube notation Singmaster, 1981):

- 3-corner swap (see Figure 5.4a): $L'\ B\ L\ F'\ L'\ B'\ L\ F$
- 3-edge swap, middle: $L'\ R\ U\ U\ R'\ L\ F\ F$
- 3-edge swap, face: $R\ R\ U\ R\ U\ R'\ U'\ R'\ U'\ R'\ U\ R'$
- 2-corner rotate: $R\ B'\ R'\ U'\ B'\ U\ F\ U'\ B\ U\ R\ B\ R'\ F'$
- R-permutation: $F\ F\ R'\ F'\ U'\ F'\ U\ F\ R\ F'\ U\ U\ F\ U\ U\ F'\ U'$
- 2-edge flip: $L\ R'\ F\ L\ R'\ D\ L\ R'\ B\ L\ R'\ U\ U\ L\ R'\ F\ L\ R'\ D\ L\ R'\ B\ L\ R'$

To generate the full set of 576 expert macro-actions, we consider 96 variations of each of the above sequences, including all 24 possible orientations, along with their inverse and mirror-flipped versions.

The learned 3-pair-swap macro in Figure 5.4b was not included with the expert macro-actions. We also provide its action sequence for completeness.

- 3-pair-swap (see Figure 5.4b): $F'\ L\ F'\ L'\ F\ F\ R\ U'\ R'\ F'\ U\ F$

C.5 Reproducibility

C.5.1 Hyperparameter Selection

In the main text, and the preceding sections of the appendix, we describe the final hyperparameters used to run the experiments. We arrived at these values by an informal hyperparameter search, and many hyperparameters never changed from their initial values.

C.5.1.1 PDDLGym Domains

For the PDDLGym domains, the simulation budget was set to 100K queries for compute reasons, as the PDDLGym simulator was slower than the domain-specific simulators. The macro-learning budgets for the PDDLGym domains were set to be comparable to the number of simulator queries needed to solve a single problem instance using greedy best-first search with the goal-count heuristic and primitive actions. The number of PDDLGym macros was chosen to be uniform across the various domains.

We ran some informal experiments with different amounts of macros to ensure that the approach was not overly sensitive to the number of macros, and found that there was no significant change in performance when

adding more macros, as long as the effect size remained low. We found that it was possible to tune the number of macros for each PDDLGym domain separately, with improved results, but felt that leaving the number of macros fixed was a more principled evaluation of our approach.

C.5.1.2 15-Puzzle

The simulation budget for 15-puzzle was set to 500K queries, although this full simulation budget was not needed since every problem was solved in fewer than that many generated states. The number of macros for 15-puzzle was set higher than for PDDLGym, to compensate for the fact that the domain-specific simulator macros are tied to specific tiles, rather than lifted like the PDDLGym macros. The numbers of random and focused macros were equal to each other, to ensure a fair comparison.

C.5.1.3 Rubik's Cube

We increased the Rubik's cube simulation budget to 2M queries to see whether the primitive-action planner could solve any problems with more planning time. The macro-learning budget for Rubik's cube was set to 1M queries to see if the total cost of learning macros and planning was low enough to justify learning macros for a single problem instance. The numbers of focused and random macros for Rubik's cube were chosen to match the number of expert macro-actions, which was itself chosen so that the expert macros could efficiently solve the Rubik's cube.

C.5.2 Computational Resources

This chapter included experiments that ran a cluster of Linux machines running either RedHat 7.7 or Debian 10, with varying hardware specifications. However, a single seed for each of the experiments can run in 30 minutes (and usually significantly less) on a MacBook Pro running macOS Mojave (10.14.6), with 2GHz i5 processor and 16GB RAM. No GPUs were used for any of the experiments.

C.5.3 Random Seeds

Random seeds were used to generate the problem instances, macro-actions, and planning results. We have attempted to make results as reproducible as possible by fixing random seeds. The commands listed in the *README* file should reproduce our results exactly. As noted in the preceding sections of this appendix, we have also saved and included the generated problem instances in the linked code repository, to allow for maximum portability.

Appendix D

Task Scoping

D.1 Results using Fast Downward's Merge and Shrink Heuristic

The full configuration we used for the merge and shrink heuristic in Table D.1 is:

```
astar(
  merge_and_shrink(
    shrink_strategy=shrink_bisimulation(greedy=false),
    merge_strategy=merge_sccs(
      order_of_sccs=topological,
      merge_selector=score_based_filtering(
        scoring_functions[goal_relevance,dfp,total_order]
      )
    ),
    label_reduction=exact(before_shrinking=true,before_merging=false),
    max_states=50k,
    threshold_before_merge=1
  )
)
```

D.2 Detailed Experimental Domain Descriptions

Below, we provide detailed descriptions of our different experimental environments. The PDDL files that were actually used to run these domains in our experiments are included with the provided code submission.

Problem	Operators		Expansions		Evaluations		Translate	Scoping	Planning Time (s)		Total Time (s)	
	Unscoped	Scoped	Unscoped	Scoped	Unscoped	Scoped			Unscoped	Scoped	Unscoped	Scoped
Driverlog 15	2,592	**2,112**	527,636	**460,244**	8,796,110	**7,289,831**	0.5 ± 0.0	3.6 ± 0.2	12.4 ± 0.5	$\mathbf{9.4 \pm 0.4}$	$\mathbf{12.9 \pm 0.5}$	13.5 ± 0.5
Driverlog 16	4,890	**3,540**	38,681	**29,618**	925,264	**577,118**	0.7 ± 0.0	8.4 ± 0.4	7.6 ± 0.2	$\mathbf{4.5 \pm 0.2}$	$\mathbf{8.3 \pm 0.3}$	13.6 ± 0.6
Driverlog 17	6,170	**3,770**	7,768,684	**4,607,258**	198,623,900	**88,705,990**	0.8 ± 0.0	9.5 ± 0.3	154.0 ± 4.8	$\mathbf{50.2 \pm 1.5}$	154.9 ± 4.9	$\mathbf{60.6 \pm 1.8}$
Logistics 15	650	**250**	5,951,997	**672,736**	140,607,200	**11,646,320**	0.3 ± 0.0	0.7 ± 0.0	73.7 ± 2.9	$\mathbf{11.1 \pm 0.8}$	74.0 ± 2.9	$\mathbf{12.1 \pm 0.8}$
Logistics 20	650	**250**	289,584	**86,663**	6,941,424	**1,524,997**	0.3 ± 0.0	0.7 ± 0.0	$\mathbf{7.8 \pm 0.4}$	8.6 ± 0.5	$\mathbf{8.1 \pm 0.4}$	9.5 ± 0.6
Logistics 25	650	**290**	11,437,240	**1,944,960**	265,871,100	**35,198,000**	0.3 ± 0.0	0.8 ± 0.0	148.0 ± 5.2	$\mathbf{21.5 \pm 1.1}$	148.3 ± 5.3	$\mathbf{22.5 \pm 1.1}$
Satellite 05	609	**339**	114	114	7,001	**3,950**	0.3 ± 0.1	0.6 ± 0.1	$\mathbf{3.9 \pm 0.2}$	4.5 ± 0.2	$\mathbf{4.2 \pm 0.2}$	5.5 ± 0.3
Satellite 06	582	**362**	21	21	1,084	**684**	0.3 ± 0.1	0.5 ± 0.1	$\mathbf{1.0 \pm 0.1}$	1.4 ± 0.1	$\mathbf{1.4 \pm 0.1}$	2.2 ± 0.1
Satellite 07	983	**587**	$> 21,423,400$	**13,438,440**	$> 362,395,000$	**679,734,000**	0.4 ± 0.1	0.9 ± 0.1	$> 813.3 \pm 4.2$	$\mathbf{203.2 \pm 5.6}$	$> 838.1 \pm 18.3$	$\mathbf{204.5 \pm 5.7}$
Zenotravel 10	1,155	**1,095**	1,466,718	**1,031,280**	37,782,140	**25,677,010**	0.3 ± 0.0	2.4 ± 0.0	22.1 ± 0.4	$\mathbf{16.6 \pm 0.3}$	22.5 ± 0.4	$\mathbf{19.3 \pm 0.4}$
Zenotravel 12	3,375	**3,159**	5,306,442	**3,348,866**	231,592,500	**140,117,800**	0.6 ± 0.0	7.3 ± 0.1	125.4 ± 4.9	$\mathbf{72.3 \pm 3.6}$	125.9 ± 5.0	$\mathbf{80.2 \pm 3.7}$
Zenotravel 14	6,700	**6,200**	$> 6,462,279$	$> 5,392,118$	$> 168,064,648$	$> 130,752,144$	1.0 ± 0.0	14.3 ± 0.3	$> 644.6 \pm 11.5$	$> 661.4 \pm 17.0$	$> 644.6 \pm 11.5$	$> 676.2 \pm 17.1$

Table D.1: Results for our Fast Downward experiments using the Merge and Shrink heuristic. Entries beginning with $>$ indicate that Fast Downward did not find a plan, due to an out-of-memory error. Note that Satellite 07 could only be completed when scoped, and that Logistics 25 was over 6 times as fast when using scoping.

D.2.1 Numeric Planning Domains

D.2.1.1 Multi-Switch Continuous Playroom

In this domain, an agent controls 3 effectors (an eye, a hand, and a marker) to interact with 6 kinds of objects (a light switch, a red button, a green button, a ball, a bell, and a monkey). The agent exists in a grid where it can take an action to move its effectors in the cardinal directions. To interact with the light switch or buttons, the agent's eye and hand effectors must be at the same grid cell as the relevant object. The light switch can be turned on and off to toggle the playroom's lights, and, when the lights are on, the green button can be pushed to turn on music, while the red button can be pushed to turn off music. Once the music is on, regardless of the state of the lights, the agent can move its eye and hand effectors to the ball and its marker effector to the bell to throw the ball at the bell and frighten the monkey. For this particular goal, if the green buttons are already pressed in the initial state, then all green buttons, as well as all light switches, are rendered task irrelevant.

In our experiments, we created tasks within progressively larger versions of the domain by progressively increasing the number of pressed green buttons and light-switches. The optimal plan for each of these tasks is for the agent to navigate its eye and hand effectors to the ball, and navigate the marker to the bell and then throw the ball at the bell to make the monkey scream.

D.2.1.2 Composite IPC Domain

This domain was constructed by simply combining the numeric *Satellites*, *Driverlog*, *Depots*, and *Zenotravel* domains from the 2002 IPC. The combination was done straightforwardly: the corresponding sections of the different domain files (i.e. `types`, `predicates`, `functions`, `operators`) were simply appended to form one combined section in the new composite domain file. To create the composite problem files, we chose one problem file from each domain and combined the objects and initial states. However, we did not combine the goals (i.e., we created four separate files that had the same composite objects and initial state, but had the goal of the corresponding original problem file from each of the four respective domains). Below, we provide a description of the dynamics of the individual domains. The dynamics of the composite domain are simply the union of those of all the individual domains.

D.2.1.2.1 Satellites In this domain, the agent controls a host of satellites, each equipped with various instruments. The various instruments can be switched on or off independently. They can also be calibrated by pointing them at specific calibration targets. The instruments can also take images of specific phenomena if they are calibrated. Finally, the satellite itself can be angled to point at specific phenomena. Repositioning satellites requires power, and taking readings uses up storage space, both of which are finite. Problems involve procuring images of specific phenomena with specific instruments and pointing specific satellites in particular directions.

D.2.1.2.2 Driverlog In the DriverLog domain, the agent must organize drivers and trucks to transport packages to specific locations. Trucks can be loaded and unloaded with packages, drivers can disembark and board other trucks, and trucks can drive only between locations that are connected. Additionally, driving or walking between locations incurs different amounts of time. Problems involve moving certain drivers, trucks and packages to specific locations with no constraints on time.

D.2.1.2.3 Depots In the Depots domain, the agent must controls various vehicles and cranes that must coordinate together to transport large containers from one location to another. Containers have weights and vehicles have defined load limits that cannot be exceeded. Problems involve ensuring particular containers are left in specific locations.

D.2.1.2.4 Zenotravel In the ZenoTravel domain, the agent must route people and airplanes to specific cities. People can board or disembark from airplanes, and planes can fly between connected cities. When flying, planes can either choose to travel at a normal speed or ‘zoom’, which consumes more fuel. However, zooming can only be done when the number of passengers on the plane is smaller than a prescribed amount. Moreover, planes can be refueled up to a defined capacity. Aircrafts can also be refueled at any location. Most problems require the agent to get various people and planes to specific cities.

D.2.1.3 Minecraft

In this domain, the agent controls Minecraft’s central playable character and can move infinitely in either the x or y directions (though all the interactable objects necessary to complete any of the agent’s tasks are located in a fairly small grid in front of the agent as pictured in Figure 1). The agent possesses a diamond axe and three "blocks" of wool in the initial state, and is standing in front of a grid of plants (white flowers, oak saplings and blue flowers). The agent can "pluck" any plant by hitting it repeatedly and then replant it at any location. There is also a set of items, namely four diamonds and seven sticks, beside the grid of plants. The agent can "pick" any of these items by moving to the same location as them, and also place them at any other location. Finally, there are two wooden blocks placed ahead of the grid of plants. Unlike the plants or items, these blocks are solid objects (like the wool blocks the agent already possesses) and will obstruct the agent’s path. However,

the agent can destroy these blocks with its axe, which them to be automatically picked up by the agent. As with any item, the agent can then place them anywhere, whereupon they will become solid objects again.

The items within the domain can be used to "craft" various other items. If the agent possesses three blue flowers (obtained by plucking), it can invoke an action to craft a blue dye. This dye can be applied to any of the wool blocks to turn them blue. The agent can also craft a diamond axe from three diamonds and two sticks. For every wooden block the agent possesses, it can choose to craft four wooden plank blocks. Finally, the agent can craft a blue bed from three blue-dyed woolen blocks and 3 planks. Note that all these crafted items (except for the diamond axe) are items and can be picked and placed at any location. The bed and wooden planks are solid object blocks that obstruct the agent's movement and must be destroyed with the axe to be picked up and moved.

Within this domain, we defined three different tasks: (1) dye three wool blocks blue, (2) mine wood using a diamond axe and use this to craft wooden planks, and (3) craft a blue bed and place it at a specific location. To complete (1), the agent must pluck the three blue flowers from the center of the grid of plants, craft blue dye, then apply the dye to the wool blocks. To complete (2), the agent must use its axe to break one of the two wooden blocks in the domain, then invoke an action to craft planks. To accomplish (3), the agent must accomplish (1) and (2), then use three dyed wool blocks and three wooden planks to craft a bed. For task (1), the diamond axe sticks and other flowers are irrelevant by backwards reachability (Section 6.2.1). For task (2), the diamond axe is causally-linked (Section 6.2.3), the flowers, sticks and diamonds are irrelevant by backwards reachability. For task (3), the wool blocks and the diamond axe are causally-linked, and all plants other than the blue flowers, as well as the diamonds and sticks are irrelevant by backwards reachability.

D.2.2 IPC Domains

Here, we describe the dynamics of each of the planning domains used for Section 6.3.2 of the main text. Note that we did not modify the dynamics of the domain whatsoever for our experiments - we only modified specific problem instances to introduce task irrelevance as described below:

D.2.2.1 Logistics

In this well-known planning domain, the agent is tasked with delivering various packages to specific destination locations. To move the packages, the agent can choose to load them into a plane and fly them between locations or load them into a truck and drive them. Trucks can drive between any locations, but airplanes can only fly between locations with airports. This problem is rather similar to the Depots domain described above in Section D.2.1.2.

We introduced irrelevance into problems by modifying the goal so that most packages were already at their goal locations in the initial state.

D.2.2.2 DriverLog

This domain is exactly the same as that described in Section D.2.1.2, except that there are no time costs incurred while driving or walking between locations.

We introduced irrelevance into problems by modifying the goal so that most conditions were already satisfied in the initial state.

D.2.2.3 Satellite

This domain is exactly the same as that described in Section D.2.1.2, except that satellites do not have power or data storage limits.

We introduced irrelevance into these problems by modifying the goal so that several of its conditions were already satisfied or almost satisfied (e.g. specific instruments were already calibrated and pointing at goal phenomena) in the initial state. We also added additional instruments and satellites but not specifying any goal conditions involving these, rendering these irrelevant as well.

D.2.2.4 ZenoTravel

This domain is exactly the same as that described in Section D.2.1.2, except there is no limit on the number of passengers that the plane can zoom with, or on the amount of fuel a plane can hold.

We introduced irrelevance into problems by modifying the goal so that many people and planes were already at their goal locations in the initial state.

D.3 Full PDDL for example domain

D.3.1 Domain File

```
(define (domain toy-example)
    (:requirements :strips)

    (:predicates
        (has-food ?ag)
        (has-sticks ?ag)
        (has-stone ?ag)
        (hungry ?ag)
        (has-axe ?ag)
    )

    (:action hunt
        :parameters (?ag)
        :precondition (and (not (has-food ?ag)) (not (hungry ?ag)))
        :effect (and (has-food ?ag))
    )

    (:action gather
        :parameters (?ag)
        :precondition (and (not (has-food ?ag)) (hungry ?ag))
        :effect (and (has-food ?ag))
    )

    (:action get_stick
        :parameters (?ag)
        :precondition (and (not (has-sticks ?ag)))
        :effect (and (has-sticks ?ag))
    )

    (:action get_stone
        :parameters (?ag)
        :precondition (and (not (has-stone ?ag)))
        :effect (and (has-stone ?ag))
    )

    (:action eat
        :parameters (?ag)
        :precondition (and (has-food ?ag) (hungry ?ag))
```

```
        :effect (and (not (has-food ?ag)) (not (hungry ?ag)))
    )

    (:action make_axe
        :parameters (?ag)
        :precondition (and (has-sticks ?ag) (has-stone ?ag) (not (has-axe ?ag)))
        :effect (and (not (has-sticks ?ag)) (not (has-stone ?ag)) (has-axe ?ag))
    )

    (:action wait
        :parameters (?ag)
        :precondition (and (not (hungry ?ag)))
        :effect (and (hungry ?ag))
    )
)
```

D.3.2 Problem File

```
(define (problem example-1)
    (:domain toy-example)

    (:objects steve)

    (:init
        (not (has-food steve))
        (not (has-sticks steve))
        (not (has-stone steve))
        (not (hungry steve))
        (not (has-axe steve))
    )

    (:goal (and (not (hungry steve)) (has-axe steve)))
)
```

Bibliography

Abel, David, Dilip Arumugam, Kavosh Asadi, Yuu Jinnai, Michael L. Littman, and Lawson L.S. Wong (2019). "State Abstraction as Compression in Apprenticeship Learning". In: *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, pp. 3134–3142.

Abel, David, Dilip Arumugam, Lucas Lehnert, and Michael L. Littman (2018). "State Abstractions for Lifelong Reinforcement Learning". In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80, pp. 10–19.

Abel, David, D. Ellis Hershkowitz, and Michael L. Littman (2016). "Near Optimal Behavior via Approximate State Abstraction". In: *Proceedings of the 33rd International Conference on Machine Learning*. Vol. 48, pp. 2915–2923.

Agostinelli, Forest, Stephen McAleer, Alexander Shmakov, and Pierre Baldi (2019). "Solving the Rubik's cube with deep reinforcement learning and search". In: *Nature Machine Intelligence* 1, pp. 356–363.

Agrawal, Pulkit, Ashvin Nair, Pieter Abbeel, Jitendra Malik, and Sergey Levine (2016). "Learning to Poke by Poking: Experiential Learning of Intuitive Physics". In: *Advances in Neural Information Processing Systems*. Vol. 29, pp. 5074–5082.

Alcázar, Vidal and Álvaro Torralba (2015). "A Reminder about the Importance of Computing and Exploiting Invariants in Planning". In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling*, pp. 2–6.

Anand, Ankesh, Evan Racah, Sherjil Ozair, Yoshua Bengio, Marc-Alexandre Côté, and R. Devon Hjelm (2019). "Unsupervised State Representation Learning in Atari". In: *Advances in Neural Information Processing Systems*. Vol. 32, pp. 8769–8782.

Asadi, Kavosh, Dipendra Misra, and Michael Littman (2018). "Lipschitz continuity in model-based reinforcement learning". In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80, pp. 264–273.

Asadi, Kavosh, Neev Parikh, Ronald E. Parr, George D. Konidaris, and Michael L. Littman (2021). "Deep Radial Basis Value Functions for Continuous Control". In: *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*, pp. 6696–6704.

Asai, Masataro and Alex Fukunaga (2015). "Solving Large-Scale Planning Problems by Decomposition and Macro Generation". In: *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling*, pp. 16–24.

Asai, Masataro and Alex Fukunaga (2018). "Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*.

Bäckström, Christer (1992). "Equivalence and Tractability Results for SAS+ Planning". In: *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*.

Bäckström, Christer and Bernhard Nebel (1995). “Complexity results for SAS+ planning”. In: *Computational Intelligence* 11.4, pp. 625–655.

Bai, Aijun, Siddharth Srivastava, and Stuart J Russell (2016). “Markovian State and Action Abstractions for MDPs via Hierarchical MCTS.” In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pp. 3029–3037.

Bakker, Bram (2001). “Reinforcement learning with long short-term memory”. In: *Advances in Neural Information Processing Systems* 14.

Bellemare, Marc G., Salvatore Candido, Pablo Samuel Castro, Jun Gong, Marlos C. Machado, Subhodeep Moitra, Sameera S. Ponda, and Ziyu Wang (2020). “Autonomous navigation of stratospheric balloons using reinforcement learning”. In: *Nature*, pp. 77–82.

Bellemare, Marc G., Yavar Naddaf, Joel Veness, and Michael Bowling (2013). “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *Journal of Artificial Intelligence Research* 47, pp. 253–279.

Bellman, R. (1954). “The Theory of Dynamic Programming”. In: *Bulletin of the American Mathematical Society* 60.6, pp. 503–515.

Biza, Ondrej, Robert Platt, Jan-Willem van de Meent, and Lawson L. S. Wong (2021). “Learning Discrete State Abstractions With Deep Variational Inference”. In: *Third Symposium on Advances in Approximate Bayesian Inference*.

Bonet, Blai and Héctor Geffner (2001). “Planning as heuristic search”. In: *Artificial Intelligence* 129, pp. 5–33.

Bonet, Blai, Gábor Loerincs, and Héctor Geffner (1997). “A robust and fast action selection mechanism for planning”. In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, pp. 714–719.

Boots, Byron, Arthur Gretton, and Geoffrey J. Gordon (2013). “Hilbert Space Embeddings of Predictive State Representations”. In: *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*, pp. 92–101.

Boots, Byron, Sajid M. Siddiqi, and Geoffrey J. Gordon (2011). “Closing the learning-planning loop with predictive state representations”. In: *The International Journal of Robotics Research* 30.7, pp. 954–966.

Botea, Adi, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer (2005). “Macro-FF: Improving AI planning with automatically learned macro-operators”. In: *Journal of Artificial Intelligence Research* 24, pp. 581–621.

Boutilier, Craig, Thomas Dean, and Steve Hanks (1999). “Decision-theoretic planning: Structural assumptions and computational leverage”. In: *Journal of Artificial Intelligence Research* 11, pp. 1–94.

Bradbury, James, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang (2018). *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13.

Brüngger, Adrian, Ambros Marzetta, Komei Fukuda, and Jurg Nievergelt (1999). “The parallel search bench ZRAM and its applications”. In: *Annals of Operations Research* 90, pp. 45–63.

Büchner, Clemens (2018). “Abstraction Heuristics for Rubik’s Cube”. Bachelor’s Thesis. University of Basel.

Burda, Yuri, Harrison Edwards, Amos J. Storkey, and Oleg Klimov (2018). “Exploration by Random Network Distillation”. In: *arXiv*, 1810.12894.

Cassandra, Anthony (2003). *The POMDP Page*. https://www.pomdp.org/. Accessed: 2023-01-14.

Cassandra, Anthony R., Leslie Pack Kaelbling, and Michael L. Littman (1994). “Acting optimally in partially observable stochastic domains”. In: *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*. Vol. 94, pp. 1023–1028.

Castro, Pablo Samuel (2020). “Scalable Methods for Computing State Similarity in Deterministic Markov Decision Processes”. In: *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence*, pp. 10069–10076.

Chen, Yixin, Zhao Xing, and Weixiong Zhang (2007). "Long-Distance Mutual Exclusion for Propositional Planning". In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, pp. 1840–1845.

Choi, Jongwook, Yijie Guo, Marcin Moczulski, Junhyuk Oh, Neal Wu, Mohammad Norouzi, and Honglak Lee (2019). "Contingency-Aware Exploration in Reinforcement Learning". In: *International Conference on Learning Representations*.

Chrisman, Lonnie (1992). "Reinforcement learning with perceptual aliasing: The perceptual distinctions approach". In: *Proceedings of the Tenth National Conference on Artificial intelligence*. Vol. 1992, pp. 183–188.

Christiano, Paul, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba (2016). "Transfer from Simulation to Real World through Learning Deep Inverse Dynamics Model". In: *arXiv*, 1610.03518.

Chrpa, Lukáš, Mauro Vallati, and Thomas Leo McCluskey (2014). "MUM: A technique for maximising the utility of macro-operators by constrained generation and use". In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, pp. 65–73.

Coles, Andrew and Amanda Smith (2007). "Marvin: A Heuristic Search Planner with Online Macro-Action Learning". In: *Journal of Artificial Intelligence Research* 28, pp. 119–156.

Corneil, Dane, Wulfram Gerstner, and Johanni Brea (2018). "Efficient Model-Based Deep Reinforcement Learning with Variational State Tabulation". In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80, pp. 1049–1058.

Culberson, Joseph C. and Jonathan Schaeffer (1998). "Pattern Databases". In: *Computational Intelligence* 14.3, pp. 318–334.

Dawson, Clive and Laurent Siklossy (1977). "The Role of Preprocessing in Problem Solving Systems: "An Ounce of Reflection is Worth a Pound of Backtracking"". In: *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pp. 465–471.

Dayan, Peter and Geoffrey E. Hinton (1992). "Feudal reinforcement learning". In: *Advances in Neural Information Processing Systems* 5.

De Moura, Leonardo and Nikolaj Bjørner (2008). "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340.

Dean, Thomas and Robert Givan (1997). "Model Minimization in Markov Decision Processes". In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pp. 106–111.

Diuk, Carlos, Lihong Li, and Bethany R. Leffler (2009). "The Adaptive k-Meteorologists Problem and Its Application to Structure Learning and Feature Selection in Reinforcement Learning". In: *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 249–256.

Domshlak, Carmel, Jörg Hoffmann, and Michael Katz (2015). "Red-black planning: A new systematic approach to partial delete relaxation". In: *Artificial Intelligence* 221, pp. 73–114.

Dornhege, Christian, Patrick Eyerich, Thomas Keller, Sebastian Trüg, Michael Brenner, and Bernhard Nebel (2012). "Semantic attachments for domain-independent planning systems". In: *Towards Service Robots for Everyday Environments: Recent Advances in Designing Service Robots for Complex Tasks in Everyday Environments*, pp. 99–115.

Downey, Carlton, Ahmed Hefny, Byron Boots, Geoffrey J Gordon, and Boyue Li (2017). "Predictive state recurrent neural networks". In: *Advances in Neural Information Processing Systems* 30.

Du, Simon S., Akshay Krishnamurthy, Nan Jiang, Alekh Agarwal, Miroslav Dudík, and John Langford (2019). "Provably efficient RL with Rich Observations via Latent State Decoding". In: *Proceedings of the 36th International Conference on Machine Learning*. Vol. 97, pp. 1665–1674.

Edelkamp, Stefan and Malte Helmert (1999). "Exhibiting Knowledge in Planning Problems to Minimize State Encoding Length". In: *European Conference on Planning*, pp. 135–147.

Ferns, Norm, Prakash Panangaden, and Doina Precup (2004). "Metrics for Finite Markov Decision Processes". In: *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*. Vol. 4, pp. 162–169.

Ferns, Norm, Prakash Panangaden, and Doina Precup (2011). "Bisimulation metrics for continuous Markov decision processes". In: *SIAM Journal on Computing* 40.6, pp. 1662–1714.

Fikes, Richard E. and Nils J. Nilsson (1970). *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*. Tech. rep. 43. Menlo Park, CA: Artificial Intelligence Group, Stanford Research Institute, pp. 1–37.

Finn, Chelsea, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel (2016). "Deep Spatial Autoencoders for Visuomotor Learning". In: *IEEE International Conference on Robotics and Automation*, pp. 512–519.

Fišer, Daniel, Álvaro Torralba, and Alexander Shleyfman (2019). "Operator Mutexes and Symmetries for Simplifying Planning Tasks". In: *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, pp. 7586–7593.

Fox, Maria and Derek Long (2003). "PDDL2.1: An extension to PDDL for expressing temporal planning domains". In: *Journal of Artificial Intelligence Research* 20, pp. 61–124.

Francès, Guillem, Miquel Ramírez, Nir Lipovetzky, and Hector Geffner (2017). "Purely Declarative Action Descriptions are Overrated: Classical Planning with Simulators". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pp. 4294–4301.

Galataud, Antoine (2021). "HVAC Processes Control: Can AI Help?" In: *Foobot Technical Blog*. URL: https://techblog.foobot.io/hvac/control/ai/reinforcement_learning/smart_control.html.

Gelada, Carles, Saurabh Kumar, Jacob Buckman, Ofir Nachum, and Marc G. Bellemare (2019). "DeepMDP: Learning Continuous Latent Space Models for Representation Learning". In: *Proceedings of the 36th International Conference on Machine Learning*. Vol. 97, pp. 2170–2179.

Gerevini, Alfonso E., Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos (2009). "Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners". In: *Artificial Intelligence* 173.5-6, pp. 619–668.

Guestrin, Carlos, Relu Patrascu, and Dale Schuurmans (2002). "Algorithm-Directed Exploration for Model-Based Reinforcement Learning in Factored MDPs". In: *Proceedings of the Nineteenth International Conference on Machine Learning*, pp. 235–242.

Gutmann, Michael and Aapo Hyvärinen (2010). "Noise-contrastive estimation: A new estimation principle for unnormalized statistical models". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 297–304.

Ha, David and Jürgen Schmidhuber (2018). "World Models". In: *arXiv*, 1803.10122.

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine (2018). "Soft actor-critic: Off-policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *Proceedings of the International Conference on Machine Learning*, pp. 1861–1870.

Hafner, Danijar, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi (2020). "Dream to Control: Learning Behaviors by Latent Imagination". In: *International Conference on Learning Representations*.

Haslum, Patrik, Malte Helmert, and Anders Jonsson (2013). "Safe, Strong, and Tractable Relevance Analysis for Planning". In: *Proceedings of the International Conference on Automated Planning and Scheduling*, pp. 317–321.

Helmert, Malte (2006). "The Fast Downward Planning System". In: *Journal of Artificial Intelligence Research* 26, pp. 191–246.

Helmert, Malte and Carmel Domshlak (2009). "Landmarks, Critical Paths and Abstractions: What's the Difference Anyway?" In: *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pp. 162–169.

Helmert, Malte, Haslum Patrik, Jörg Hoffmann, and Raz Nissim (2014). "Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces". In: *Journal of the ACM* 61, 16:1–16:63.

Helmert, Malte, Patrik Haslum, Jörg Hoffmann, et al. (2007). "Flexible abstraction heuristics for optimal sequential planning". In: *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*. AAAI Press.

Hester, Todd and Peter Stone (2012). "Learning and Using Models". In: *Reinforcement Learning: State of the Art*. Springer, pp. 111–141.

Higgins, Irina, Arka Pal, Andrei Rusu, Loic Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, and Alexander Lerchner (2017). "DARLA: Improving Zero-Shot Transfer in Reinforcement Learning". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70, pp. 1480–1490.

Hoffmann, Jörg and Bernhard Nebel (2001). "The FF planning system: Fast plan generation through heuristic search". In: *Journal of Artificial Intelligence Research* 14, pp. 253–302.

Hoffmann, Jörg, Ashish Sabharwal, and Carmel Domshlak (2006). "Friends or Foes? An AI Planning Perspective on Abstraction and Search". In: *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, pp. 294–303.

Horčík, Rostislav and Daniel Fišer (2021). "Endomorphisms of Classical Planning Tasks". In: *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*, pp. 11835–11843.

Hutter, Marcus (2016). "Extreme state aggregation beyond Markov decision processes". In: *Theoretical Computer Science* 650, pp. 73–91.

Jinnai, Yuu and Alex Fukunaga (2017). "Learning to prune dominated action sequences in online black-box planning". In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pp. 839–845.

Jonschkowski, Rico and Oliver Brock (2015). "Learning state representations with robotic priors". In: *Autonomous Robots* 39.3, pp. 407–428.

Kaelbling, Leslie Pack, Michael L Littman, and Anthony R Cassandra (1995). "Partially observable markov decision processes for artificial intelligence". In: *International Workshop on Reasoning with Uncertainty in Robotics*, pp. 146–163.

Kaelbling, Leslie Pack, Michael L. Littman, and Anthony R. Cassandra (1998). "Planning and acting in partially observable stochastic domains". In: *Artificial Intelligence* 101.1-2, pp. 99–134.

Kaiser, Łukasz, Mohammad Babaeizadeh, Piotr Miłos, Błażej Osiński, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski (2020). "Model Based Reinforcement Learning for Atari". In: *International Conference on Learning Representations*.

Kapturowski, Steven, Georg Ostrovski, John Quan, Rémi Munos, and Will Dabney (2018). "Recurrent Experience Replay in Distributed Reinforcement Learning". In: *International Conference on Learning Representations*.

Katz, Michael and Carmel Domshlak (2010). "Implicit Abstraction Heuristics". In: *Journal of Artificial Intelligence Research* 39, pp. 51–126.

Kearns, Michael and Daphne Koller (1999). "Efficient Reinforcement Learning in Factored MDPs". In: *Proceedings of the 16th International Joint Conference on Artificial Intelligence*. Vol. 2, pp. 740–747.

Keyder, Emil, Jörg Hoffmann, and Patrik Haslum (2014). "Improving Delete Relaxation Heuristics Through Explicitly Represented Conjunctions". In: *Journal of Artificial Intelligence Research* 50, pp. 487–533.

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *Proceedings of the 3rd International Conference on Learning Representations*.

Koehler, Jana and Kilian Schuster (2000). "Elevator Control as a Planning Problem." In: *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pp. 331–338.

Koller, Daphne and Ronald Parr (1999). "Computing factored value functions for policies in structured MDPs". In: *Proceedings of the 16th International Joint Conference on Artificial Intelligence*. Vol. 99, pp. 1332–1339.

Konidaris, George (2019). "On the necessity of abstraction". In: *Current Opinion in Behavioral Sciences* 29, pp. 1–7.

Konidaris, George, Leslie Pack Kaelbling, and Tomas Lozano-Perez (2018). "From Skills to Symbols: Learning Symbolic Representations for Abstract High-Level Planning". In: *Journal of Artificial Intelligence Research* 61, pp. 215–289.

Korf, Richard E. (1985a). "Depth-first iterative-deepening: An optimal admissible tree search". In: *Artificial Intelligence* 27.1, pp. 97–109.

Korf, Richard E. (1985b). "Macro-Operators: A Weak Method for Learning". In: *Artificial Intelligence* 26.1, pp. 35–77.

Krantz, Steven G. and Harold R. Parks (2002). *A Primer of Real Analytic Functions*. 2nd ed. Birkhäuser.

Kurutach, Thanard, Aviv Tamar, Ge Yang, Stuart Russell, and Pieter Abbeel (2018). "Learning Plannable Representations with Causal InfoGAN". In: *Advances in Neural Information Processing Systems*. Vol. 31, pp. 8747–8758.

Kushmerick, Nicholas, Steve Hanks, and Daniel S Weld (1995). "An algorithm for probabilistic planning". In: *Artificial Intelligence* 76.1-2, pp. 239–286.

Laskin, Michael, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas (2020a). "Reinforcement Learning with Augmented Data". In: *Advances in Neural Information Processing Systems*. Vol. 33, pp. 19884–19895.

Laskin, Michael, Aravind Srinivas, and Pieter Abbeel (2020b). "CURL: Contrastive Unsupervised Representations for Reinforcement Learning". In: *Proceedings of the 37th International Conference on Machine Learning*. Vol. 119, pp. 5639–5650.

Lee, Alex X., Anusha Nagabandi, Pieter Abbeel, and Sergey Levine (2020). "Stochastic Latent Actor-Critic: Deep Reinforcement Learning with a Latent Variable Model". In: *Advances in Neural Information Processing Systems*. Vol. 33, pp. 741–752.

Lehnert, Lucas and Michael L Littman (2020). "Successor Features Combine Elements of Model-free and Model-based Reinforcement Learning". In: *Journal of Machine Learning Research* 21.196, pp. 1–53.

Li, Lihong, Thomas J. Walsh, and Michael L. Littman (2006). "Towards a Unified Theory of State Abstraction for MDPs." In: *International Symposium on Artificial Intelligence and Mathematics*.

Lin, Long-Ji (1992). "Self-improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching". In: *Machine Learning* 8.3, pp. 293–321.

Lin, Long-Ji and Tom M Mitchell (1993). "Reinforcement learning with hidden states". In: *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pp. 271–280.

Lipovetzky, Nir and Hector Geffner (2012). "Width and serialization of classical planning problems". In: *Proceedings of the 20th European Conference on Artificial Intelligence*, pp. 540–545.

Lipovetzky, Nir and Hector Geffner (2017). "Best-First Width Search: Exploration and Exploitation in Classical Planning". In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pp. 3590–3596.

Lipovetzky, Nir, Miquel Ramirez, and Hector Geffner (2015). "Classical planning with simulators: Results on the Atari video games". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pp. 1610–1616.

Littman, Michael L., Anthony R. Cassandra, and Leslie Pack Kaelbling (1995). "Learning policies for partially observable environments: Scaling up". In: *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 362–370.

Littman, Michael L., Richard S. Sutton, and Satinder Singh (2001). "Predictive Representations of State". In: *Advances in Neural Information Processing Systems*. Vol. 14.

Long, Derek and Maria Fox (2003). "The 3rd International Planning Competition: Results and Analysis". In: *Journal of Artificial Intelligence Research* 20, pp. 1–59.

Long, Derek, Henry Kautz, Bart Selman, Blai Bonet, Hector Geffner, Jana Koehler, Michael Brenner, Joerg Hoffmann, Frank Rittinger, Corin R Anderson, et al. (2000). "The AIPS-98 planning competition". In: *AI Magazine* 21.2, pp. 13–13.

Markov, Andrey Andreyevich (1906). "Extension of the law of large numbers to dependent quantities". In: *[Russian] Izv. Fiz.-Matem. Obsch. Kazan Univ. (2nd Ser)* 15.1, pp. 135–156.

Matloob, Rami and Mikhail Soutchanski (2016). "Exploring Organic Synthesis with State-of-the-Art Planning Techniques". In: *ICAPS Scheduling and Planning Applications woRKshop (SPARK)*.

Mattner, Jan, Sascha Lange, and Martin Riedmiller (2012). "Learn to Swing Up and Balance a Real Pole Based on Raw Visual Input Data". In: *Neural Information Processimng*, pp. 126–133.

McAllester, David A. and David Rosenblitt (1991). "Systematic Nonlinear Planning". In: *Proceedings of the Ninth National Conference on Artificial Intelligence*.

McDermott, Drew, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins (1998). *PDDL — The Planning Domain Definition Language*. Tech. rep. 1165 (CVC Report 98-003). Yale Center for Computational Vision and Control, pp. 1–26.

Misra, Dipendra, Mikael Henaff, Akshay Krishnamurthy, and John Langford (2020). "Kinematic State Abstraction and Provably Efficient Rich-Observation Reinforcement Learning". In: *Proceedings of the 37th International Conference on Machine Learning*. Vol. 119, pp. 6961–6971.

Mityagin, Boris Samuilovich (2020). "The zero set of a real analytic function". In: *Matematicheskie Zametki* 107.3, pp. 473–475.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Nebel, Bernhard, Yannis Dimopoulos, and Jana Koehler (1997). "Ignoring Irrelevant Facts and Operators in Plan Generation". In: *Proceedings of the 4th European Conference on Planning*, pp. 338–350.

Pathak, Deepak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell (2017). "Curiosity-driven Exploration by Self-supervised Prediction". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70, pp. 2778–2787.

Pazis, Jason and Ronald Parr (2013). "PAC Optimal Exploration in Continuous Space Markov Decision Processes". In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, pp. 774–781.

Pirotta, Matteo, Marcello Restelli, and Luca Bascetta (2015). "Policy gradient in Lipschitz Markov Decision Processes". In: *Machine Learning* 100.2-3, pp. 255–283.

Pommerening, Florian, Malte Helmert, Gabriele Röger, and Jendrik Seipp (2015). "From Non-Negative to General Operator Cost Partitioning". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pp. 3335–3341.

Puterman, Martin L. (1994). *Markov Decision Processes*. Wiley.

Ravindran, Balaraman and Andrew G. Barto (2004). "Approximate Homomorphisms: A Framework for Non-exact Minimization in Markov Decision Processes". In: *Proceedings of the Fifth International Conference on Knowledge Based Computer Systems*.

Refanidis, Ioannis, Nick Bassiliades, I. Vlahavas, and Thessaloniki Greece (2001). "AI Planning For Transportation Logistics". In: *Proceedings of the 36th Annual International Logistics Conference and Exhibition (SOLE)*.

Richter, Silvia and Matthias Westphal (2010). "The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks". In: *Journal of Artificial Intelligence Research* 39, pp. 127–177.

Rokicki, Tomas (2014). *God's Number is 26 in the Quarter-turn Metric*. http://www.cube20.org/qtm/. [Online; accessed 16-April-2021].

Rovner, Alexander, Silvan Sievers, and Malte Helmert (2019). "Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning". In: *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling*, pp. 362–367.

Rummery, Gavin A. and Mahesan Niranjan (1994). *On-Line Q-learning using connectionist systems*. Tech. rep. 166. Cambridge University Engineering Department, pp. 1–20.

Russell, Stuart and Peter Norvig (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Sandewall, Erik and Ralph Rönnquist (1986). "A Representation of Action Structures". In: *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*.

Sawada, Yoshihide, Luca Rigazio, Koji Morikawa, Masahiro Iwasaki, and Yoshua Bengio (2018). "Disentangling Controllable and Uncontrollable Factors by Interacting with the World". In: *NeurIPS Workshop on Deep Reinforcement Learning*.

Scala, Enrico, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramírez (2016a). "Interval-Based Relaxation for General Numeric Planning". In: *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, pp. 655–663.

Scala, Enrico, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramírez (2020). "Subgoaling Techniques for Satisficing and Optimal Numeric Planning". In: *Journal of Artificial Intelligence Research* 68, pp. 691–752.

Scala, Enrico, Patrik Haslum, Sylvie Thiébaux, et al. (2016b). "Heuristics for numeric planning via subgoaling". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*.

Schlegel, Matthew, Andrew Jacobsen, Zaheer Abbas, Andrew Patterson, Adam White, and Martha White (2021). "General value function networks". In: *Journal of Artificial Intelligence Research* 70, pp. 497–543.

Seipp, Jendrik and Malte Helmert (2018). "Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning". In: *Journal of Artificial Intelligence Research* 62.1, pp. 535–577.

Seneta, Eugene (1996). "Markov and the birth of chain dependence theory". In: *International Statistical Review/Revue Internationale de Statistique*, pp. 255–263.

Shelhamer, Evan, Parsa Mahmoudieh, Max Argus, and Trevor Darrell (2016). "Loss is its own Reward: Self-Supervision for Reinforcement Learning". In: *arXiv*, 1612.07307.

Silver, Tom and Rohan Chitnis (2020). "PDDLGym: Gym Environments from PDDL Problems". In: *ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*. URL: https://github.com/tomsilver/pddlgym.

Silver, Tom, Rohan Chitnis, Aidan Curtis, Joshua Tenenbaum, Tomas Lozano-Perez, and Leslie Pack Kaelbling (2021). "Planning with Learned Object Importance in Large Problem Instances using Graph Neural Networks". In: *The Thirty-Fifth AAAI Conference on Artificial Intelligence*, pp. 11962–11971.

Singh, Satinder, Andrew G. Barto, and Nuttapong Chentanez (2004a). "Intrinsically motivated reinforcement learning". In: *Proceedings of the 17th International Conference on Neural Information Processing Systems*, pp. 1281–1288.

Singh, Satinder, Michael R. James, and Matthew R. Rudary (2004b). "Predictive State Representations: A New Theory for Modeling Dynamical Systems". In: *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, pp. 512–519.

Singh, Satinder, Michael L. Littman, Nicholas K. Jong, David Pardoe, and Peter Stone (2003). "Learning Predictive State Representations". In: *Proceedings of the 20th International Conference on Machine Learning*, pp. 712–719.

Singmaster, David (1981). *Notes on Rubik's magic cube*. Enslow Publishers.

Smith, Trey and Reid Simmons (2004). "Heuristic Search Value Iteration for POMDPs". In: *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*.

Song, Zhao, Ronald Parr, Xuejun Liao, and Lawrence Carin (2016). "Linear feature encoding for reinforcement learning". In: *Advances in Neural Information Processing Systems*. Vol. 29, pp. 4224–4232.

Speedsolving Wiki (2021). *CFOP method*. https://www.speedsolving.com/wiki/index.php/CFOP_method. [Online; accessed 16-April-2021].

Stooke, Adam, Kimin Lee, Pieter Abbeel, and Michael Laskin (2020). "Decoupling Representation Learning from Reinforcement Learning". In: *arXiv*, 2009.08319.

Strehl, Alexander L., Carlos Diuk, and Michael L. Littman (2007). "Efficient Structure Learning in Factored-state MDPs". In: *Proceedings of the 22nd National Conference on Artificial Intelligence*. Vol. 1, pp. 645–650.

Sutton, Richard S. (1988). "Learning to predict by the methods of temporal differences". In: *Machine Learning* 3.1, pp. 9–44.

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press.

Sutton, Richard S., Joseph Modayil, Michael Delp, Thomas Degris, Patrick M. Pilarski, Adam White, and Doina Precup (2011). "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction". In: *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pp. 761–768.

Sutton, Richard S., Doina Precup, and Satinder Singh (1999). "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial Intelligence* 112.1-2, pp. 181–211.

Tassa, Yuval, Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, and Nicolas Heess (2020). "dm_control: Software and Tasks for Continuous Control". In: *arXiv*, 2006.12983.

Thomas, Valentin, Jules Pondard, Emmanuel Bengio, Marc Sarfati, Philippe Beaudoin, Marie-Jean Meurs, Joelle Pineau, Doina Precup, and Yoshua Bengio (2017). "Independently Controllable Factors". In: *arXiv*, 1708.01289.

Thrun, Sebastian, Wolfram Burgard, and Dieter Fox (2005). *Probabilistic Robotics*. MIT Press.

Tiao, Louis (2017). *A Simple Illustration of Density Ratio Estimation and KL Divergence Estimation by Probabilistic Classification*. http://louistiao.me/notes/a-simple-illustration-of-density-ratio-estimation-and-kl-divergence-estimation-by-probabilistic-classification. [Online; accessed 16-March-2020].

Torralba, Alvaro and Jörg Hoffmann (2015). "Simulation-based admissible dominance pruning". In: *IJCAI*, pp. 1689–1695.

Torralba, Álvaro and Peter Kissmann (2015). "Focusing on What Really Matters: Irrelevance Pruning in Merge-and-Shrink". In: *SOCS*, pp. 122–130.

Vallati, Mauro and Lukáš Chrpa (2019). "On the Robustness of Domain-Independent Planning Engines: The Impact of Poorly-Engineered Knowledge". In: *Proceedings of the 10th International Conference on Knowledge Capture*, pp. 197–204.

Vallati, Mauro, Lukas Chrpa, Marek Grześ, Thomas Leo McCluskey, Mark Roberts, Scott Sanner, et al. (2015). "The 2014 international planning competition: Progress and trends". In: *AI Magazine* 36.3, pp. 90–98.

van den Oord, Aäron, Yazhe Li, and Oriol Vinyals (2018). "Representation learning with contrastive predictive coding". In: *arXiv*, 1807.03748.

van der Pol, Elise, Thomas Kipf, Frans A. Oliehoek, and Max Welling (2020). "Plannable Approximations to MDP Homomorphisms: Equivariance under Actions". In: *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems*, pp. 1431–1439.

van Hasselt, Hado, Matteo Hessel, and John Aslanides (2019). "When to use parametric models in reinforcement learning?" In: *Advances in Neural Information Processing Systems*. Vol. 32, pp. 14322–14333.

Watter, Manuel, Jost Springenberg, Joschka Boedecker, and Martin Riedmiller (2015). "Embed to control: A locally linear latent dynamics model for control from raw images". In: *Advances in Neural Information Processing Systems*. Vol. 28, pp. 2746–2754.

Werbos, Paul J. (1990). "Backpropagation Through Time: What It Does and How to Do It". In: *Proceedings of the Institute of Electrical and Electronics Engineers* 78, pp. 1550–1560.

Wilt, Christopher and Wheeler Ruml (2015). "Building a Heuristic for Greedy Search". In: *Proceedings of the Eighth International Symposium on Combinatorial Search*, pp. 131–139.

Yarats, Denis, Rob Fergus, Alessandro Lazaric, and Lerrel Pinto (2021). "Reinforcement Learning with Prototypical Representations". In: *arXiv*, 2102.11271.

Yarats, Denis and Ilya Kostrikov (2020). *Soft Actor-Critic (SAC) implementation in PyTorch*. https://github.com/denisyarats/pytorch_sac.

Yarats, Denis, Amy Zhang, Ilya Kostrikov, Brandon Amos, Joelle Pineau, and Rob Fergus (2019). "Improving Sample Efficiency in Model-Free Reinforcement Learning from Images". In: *arXiv*, 1910.01741.

Zhang, Amy, Rowan Thomas McAllister, Roberto Calandra, Yarin Gal, and Sergey Levine (2021). "Learning Invariant Representations for Reinforcement Learning without Reconstruction". In: *International Conference on Learning Representations*.

Zhang, Amy, Harsh Satija, and Joelle Pineau (2018). "Decoupling dynamics and reward for transfer learning". In: *arXiv*, 1804.10689.

Zhang, Zongzhang, Xiaoping Chen, and Michael Littman (2012). "Covering Number as a Complexity Measure for POMDP Planning and Learning". In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, pp. 1853–1859.