Reasoning about MDPs Abstractly: Bayesian Policy Search with Uncertain Prior Knowledge

Master's Thesis

Jord Molhoek

Reasoning about MDPs Abstractly: Bayesian Policy Search with Uncertain Prior Knowledge

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jord Molhoek
born in Delft, the Netherlands

Interactive Intelligence Research Group, Algorithmics Research Group
Department of Intelligent Systems, Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.tudelft.nl/ewi

© 2024 Jord Molhoek.

Cover art generated by DALL·E 2; color-spectrum and shape edited by Jord Molhoek.

Reasoning about MDPs Abstractly: Bayesian Policy Search with Uncertain Prior Knowledge

Author: Jord Molhoek
Student id: 4932919

Thesis Defense Date: February 1, 2024

Abstract

Many real-world problems fall in the category of sequential decision-making under uncertainty; Markov Decision Processes (MDPs) are a common method for modeling such problems. To solve an MDP, one could start from scratch or one could already have an idea of what good policies look like. Furthermore, there could be uncertainty in this idea. In existing literature, a policy search procedure is accelerated by encoding this prior knowledge in an action distribution which is used for policy sampling. Moreover, this is then extended by inferring these action distributions while inferring the policy through Gibbs sampling. Implicitly, this approach assumes a generalization of good and bad actions over the entire state space.

This thesis extends the existing method by leveraging a division of the state space into regions and inferring action distributions over these regions, rather than over the entire state space. We show that this can accelerate the policy search. We also show that the algorithm manages to recover if the division is unjustified. The division into regions can hence also be considered a form of prior knowledge of the policy with uncertainty. Finally, inference of the regions themselves is also explored and yields promising results.

Thesis Committee:

Thesis advisor: Dr. F. A. Oliehoek, Faculty EEMCS, TU Delft
Daily supervisor: Dr. S. Dumančić, Faculty EEMCS, TU Delft

Preface

This thesis lies on the intersection of probabilistic programming and sequential decision-making under uncertainty. Throughout the 30 weeks I worked on this thesis, I have learned a lot, not only about technical details of probabilistic programming and sequential decision-making but also about performing extensive academic research. In this thesis, I delve into the theory of sequential decision-making under uncertainty, examining specifically how some prior idea of a solution can be leveraged to find a solution more quickly. Additionally, this prior idea can have some uncertainty as it may turn out that it is unjustified. I also extend this to automatically inferring abstract representations of the underlying problem, represented as a decision tree.

Ever since I learned about automatically learning decision trees, I have been fascinated by them. Specifically, I love that anyone can put their finger on the start node and follow the lines to understand the logic. I believe that such white-box models will become more and more essential as humans realize that decisions made by Artificial Intelligence often need to be understandable, making it safe and effective to use Artificial Intelligence as a tool without completely giving up control. Meanwhile, I also love the expressive freedom that probabilistic programs provide. Again, generative models presented as human-readable code can provide meaningful insight into why an agent made a decision. Moreover, I think that Probabilistic Programming is a promising scientific field, and I can't wait to see what the coming years will bring.

I would like to express my gratitude to my supervisors Sebastijan Dumančić and Frans Oliehoek for their support and feedback throughout this project. Next, I want to thank the TU Delft PONY (PrObabilistic Neurosymbolic sYnthesis) lab for all the interesting meetings and discussions. Finally, I would like to thank my family and friends for their support throughout the journey of my Bachelor's and Master's degrees.

Jord Molhoek
Delft, the Netherlands
January 24, 2024

Contents

Preface **iii**

Contents **v**

1 Introduction **1**

2 Preliminaries **5**
2.1 Markov Decision Processes 5
2.2 Dynamic Programming for Solving MDPs 7
2.3 Probabilistic Programming 8
2.4 Metropolis-Hastings 12
2.5 Bayesian Policy Search with Policy Priors 12

3 Related Work **15**
3.1 Planning as Inference 15
3.2 Planning by Probabilistic Programming 16
3.3 Leveraging Prior Policy Knowledge 17
3.4 State Abstraction for MDPs 18

4 Methodology **19**
4.1 Problem Setting 19
4.2 The Approach: Priors over Regions 19
4.3 Generative Model 23
4.4 Inferring Policies 26
4.5 Inferring Region Configurations 29

5 Experimental Evaluation **33**
5.1 MDP Instances 33
5.2 Policy Inference Given a Region Configuration 35
5.3 Generalization of θ 44
5.4 Inference of Region Configurations 45

6 Conclusions and Future Work **51**
6.1 Conclusions and Limitations 51
6.2 Future Work 52
6.3 Broader Implications 54

Bibliography **55**

A Glossary **61**

B Hyperparameter Analysis **63**

Chapter 1

Introduction

Many real-world problems fall in the category of sequential decision-making under uncertainty; Markov Decision Processes (MDPs)[31] are a common method for modeling such problems. For example, imagine an agent in a two-dimensional maze where the goal is to move to the top (Figure 1.1 (left)). A solution to an MDP is a policy, often denoted as π. A policy is essentially a description of the behavior of an Artificially Intelligent Agent (AIA), mapping the possible situations the agent could be in (referred to as 'states') to appropriate actions. One could think of a policy as a treasure map, specifying exactly what to do in which state (Figure 1.1 (middle)). Many methods for finding a policy exist. If a policy is found by interacting with and learning from the environment, this is known as Reinforcement Learning[33]. Meanwhile, if the dynamics of the environment are known beforehand, finding a policy is known as planning[31]. This thesis focuses on Bayesian policy search: a search for a good - or even optimal - policy in the space of policies, making use of Bayesian inference (Section 2.5). This can be seen as either planning or reinforcement learning based on whether a policy is evaluated using a given model of the environment, or using interactions with the environment respectively.

To find a good policy, one could start from scratch or one could already have an idea of what good policies look like. In other words, one could have prior knowledge of good policies. Note that there is an important difference between prior knowledge of the environment and prior knowledge of the policy. In planning problems, the full dynamics of the environment are known. Prior knowledge of good policies relates to the shape or contents of a policy, or it could be extra information we have about the environment that is useful towards a certain goal. For example, we know that an agent does not have to care about the color of their hat when their goal is to exit a maze. If such knowledge is very well-defined, leveraging such knowledge is trivial; e.g. certain irrelevant state factors or irrelevant actions can be ignored, or certain state-to-action mappings can be fixed before the policy search is started. Occasionally, the prior knowledge of the policy is not as well-defined. In other words: there can be some uncertainty in the knowledge. From now on, we will refer to prior knowledge of the policy with uncertainty as *soft prior knowledge*.

In the previous example (Figure 1.1 (left)), one might have the soft prior knowledge that a good policy has a preference for the action $\uparrow$. In existing literature, such as the work of Wingate et al.[39], such knowledge is leveraged in the policy search. Specifically, policies

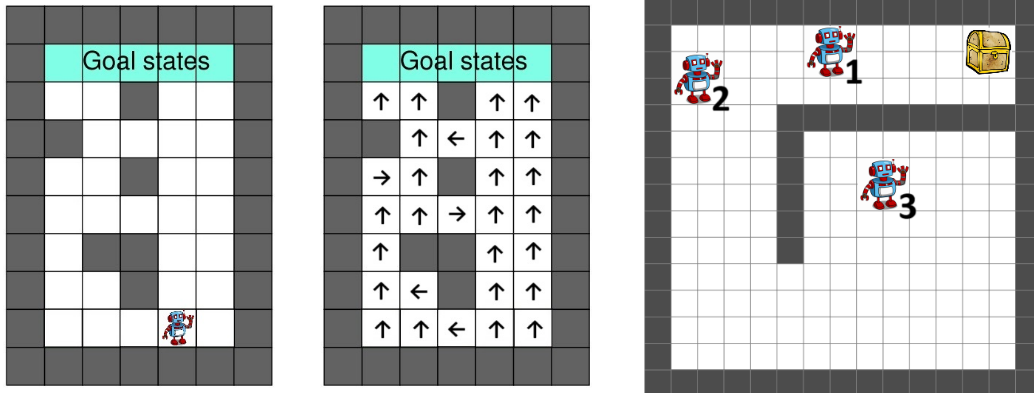

Figure 1.1: *Left:* an example grid-world where the goal is to move to the top row. *Middle:* an example policy corresponding to the left figure; mapping each state to an appropriate action. *Right:* another example of a grid-world that can be modeled as an MDP.

that contain the action $\uparrow$ often are made more probable. Next, their work aims to accelerate the search without having to specify which action should be preferred. This is done by optimizing the policy *while* optimizing the distribution of actions. Intuitively, learning that $\uparrow$ is a good action in e.g. the upper part of the maze will make $\uparrow$ also more likely in the lower part of the maze. A similar internal generalization can also work for actions that are learned to be bad.

This internal generalization of good and bad actions works well for the example from Figure 1.1 (left)[39]. However, it might not be the case in another environment (Figure 1.1 (right)). In this case, moving to the right would move agent 1 closer to the treasure. The information that moving right is a good action also generalizes to the location of agent 2. Nevertheless, this information would not generalize to the location of agent 3. There, the agent would first have to move to the bottom left to walk around the wall. This example illustrates how knowledge of good and bad actions can generalize within parts of the state space, while not generalizing over the state space entirely. Being able to identify subsets of states in which we suspect good and bad actions to generalize can be interpreted as another form of soft prior knowledge.

The aforementioned aspects motivate the main research question of this thesis, which reads:

> How can we improve Bayesian policy search by incorporating soft prior information on good policies?

Other questions that are explored are: "how do we define and represent this soft prior knowledge?", "how do we leverage it in the inference procedure?" and "how can we also (partially) infer this soft prior knowledge for generalization purposes?"

As a direct result of this thesis, the Bayesian policy search method from Wingate et al.[39] is improved by leveraging a division of the state space into regions and inferring action distributions over these regions, rather than over the entire state space. We show

that this can accelerate the policy search. We also show that the algorithm manages to recover if the division is unjustified. The division into regions can hence also be considered a form of prior knowledge of the policy with uncertainty. Finally, inference of the regions themselves is also explored and yields promising results.

The scope of this thesis is limited to MDPs with factored discrete states and un-factored discrete actions. For more information on factored MDPs, see Section 2.1 and the work of Boutilier et al.[5].

Possible broader implications of this thesis include a better understanding between humans and AI, and artificially intelligent agents (AIAs) that think more like humans. Inferring the relevant regions within the state space with a certain goal in mind yields an abstract representation of the problem. Equipping AIAs with the capability to reason about problems in an abstract way brings them closer to humans. Furthermore, by pushing the limits of this work, policies could be represented as decision trees; making the behavior of AIAs more interpretable and understandable for humans[25].

Chapter 2

Preliminaries

This chapter provides the background material for this thesis. Markov Decision Processes and relevant related concepts are explained in Sections 2.1 and 2.2. Next, Section 2.3 explains the basics of Probabilistic Programming, and Section 2.4 explains the Metropolis-Hastings algorithm. Finally, the relevant parts of the paper “Bayesian Policy Search with Policy Priors” by Wingate et al.[39] are explained in detail in Section 2.5, as this thesis builds on top of their work.

2.1 Markov Decision Processes

In sequential decision problems, an agent needs to make decisions in some environment, and the decision at one time step potentially influences the state of the (agent in the) environment in the next time step. Markov Decision Processes (MDPs) are commonly used to model such sequential decision problems. This thesis assumes the definitions of MDPs adapted from Kaelbling et al.[18]. An MDP is a tuple $(\mathcal{S},\mathcal{A},\rho,T,R)$ where

- $\mathcal{S}$ is the set of states;
- $\mathcal{A}$ is the set of actions;
- $T : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$ is the transition model. I.e. $p(s'|s,a)$ represents the probability of transitioning from state $s \in \mathcal{S}$ to state $s' \in \mathcal{S}$ after taking action $a \in \mathcal{A}$;
- $\rho : \Pi(\mathcal{S})$ is the distribution of the initial state;
- $R : \mathcal{S} \times \mathcal{A} \to \Pi(\mathbb{R})$ is the reward distribution. I.e. $R(s,a)$ is the (possibly stochastic) reward the agent receives after taking action a in state s. Occasionally, rather than mapping a (s,a)-tuple to a reward, the reward distribution is represented as $R(s)$ (e.g. by Russell and Norvig[31]) or as $R(s,a,s')$ (e.g. by Rodriguez-Sanchez et al.[27]). $\bar{R}(s,a)$ is often used to abbreviate $\mathbb{E}[R(s,a)]$.

Note that $\Pi(\cdot)$ indicates a probability distribution over the given set. Additionally, some states in $\mathcal{S}$ can be *terminal states*, which end trajectories automatically.

States and actions can be continuous or discrete. Occasionally, factored representations are used [5]. This means that a state $s \in \mathcal{S}$ can be represented as a set of variables. For example, the state of an agent in a maze with a door could be represented as $(x_coordinate, y_coordinate, has_key) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{B}$. Such factored representations can again be continuous or discrete. Additionally, a factored representation can be hybrid, meaning that there is at least one discrete state factor and at least one continuous state factor. In theory, actions can also be factored in the same way. However, for simplicity, this thesis assumes factored discrete states and un-factored discrete actions.

In the example grid-world from before (Figure 1.1 (left)), $\mathcal{S}$ would be the set of all grid cells that are not walls, and states would be represented as (x,y)-coordinates. $\mathcal{A}$ would be the set of actions that the agent could take: $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$. Furthermore, the transition model would specify that e.g. the action $\uparrow$ has a large probability of increasing the y-coordinate by one but also has a small probability of adding or subtracting one from the x-coordinate, representing a small probability that the agent could 'slip'. Moreover, a possible initial state distribution specifies that the agent has an equal probability of starting at any of the states on the bottom row: $\rho((2,2)) = \rho((3,2)) = \rho((4,2)) = \rho((5,2)) = \rho((6,2)) = 0.2$. Finally, the reward model $R(s,a)$ would specify that in any of the blue states, upon taking any action, a large reward is achieved. For more examples of MDPs, see Sections 4.2.1 and 5.1.

An MDP adheres to the (first-order) *Markov property*. This means that the next state is conditionally independent of all preceding states and actions, given the previous state and action. In mathematical terms, we can say:

$$p(s_{t+1}|s_t, s_{t-1}, ..., s_0, a_t, a_{t-1}, ..., a_0) = p(s_{t+1}|s_t, a_t) \tag{2.1}$$

A sequence of states and actions is known as a *rollout* or a *trajectory*, usually denoted by τ. Up to *horizon H* (in this thesis, H is infinite) we can say that $\tau = (s_0, a_0, s_1, a_1, ..., s_H, a_H)$. Since each (s_t, a_t)-tuple is associated with some reward, some reward value can also be attributed to a trajectory. The reward of a trajectory is often called the *return*. The returns can be additive or discounted [31]. Discounted returns are calculated as

$$R(\tau) = \sum_{t=0}^{H} \gamma^t R(s_t, a_t), \tag{2.2}$$

where $\gamma \in [0,1]$ is known as the *discount factor*. It indicates the trade-off between immediate rewards and future rewards. The case of additive return is a special case of the discounted return, i.e. when $\gamma = 1$. Moreover, since γ is a fixed parameter and the expectation of a sum is the sum of expectations we can say that

$$\bar{R}(\tau) = \sum_{t=0}^{H} \gamma^t \bar{R}(s_t, a_t) \tag{2.3}$$

A solution to an MDP is known as a policy, which is typically denoted as a mapping $\pi : \mathcal{S} \rightarrow \mathcal{A}$ as a deterministic policy or $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0,1]$ as a stochastic policy. When the dynamics of the environment are known, finding such a policy is known as *planning*. Likewise, when the agent is tasked with finding a policy without (complete) knowledge of the

dynamics of the environment, i.e. the agent needs to explore, this is known as *reinforcement learning*. The value of such a policy can be calculated as the expected return of the policy:

$$J_\pi = \int \bar{R}(\tau) p_\pi(\tau) d\tau \tag{2.4}$$

where $p_\pi(\tau)$ is the probability of observing trajectory τ given that we follow the policy π. I.e.:

$$p_\pi(\tau) = p(s_0) \prod_{t=0}^{H} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t) \tag{2.5}$$

2.2 Dynamic Programming for Solving MDPs

Finding the optimal policy for a Markov Decision Process in a planning setting can be done using dynamic programming. A popular algorithm is Value Iteration. Before Value Iteration can be understood, some common functions need to be defined. First of all, $V^\pi(s)$ describes the value of being in state s and following policy π from there. Formally this can be denoted as[1]:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \middle| s_0 = s, a_t = \pi(s_t) \right] && (2.6) \\ &= \bar{R}(s, \pi(s)) + \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^t R(s_t, \pi(s_t)) \right] && (2.7) \\ &= \bar{R}(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, \pi(s)) V^\pi(s') && (2.8) \end{aligned}$$

By extension, $Q^\pi(s,a)$ describes the value of being in state s, taking action a, and following policy π afterwards. This is formally denoted as:

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \middle| s_0 = s, a_0 = a, a_t = \pi(s_t) \right] && (2.9) \\ &= \bar{R}(s,a) + \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^t R(s_t, \pi(s_t)) \right] && (2.10) \\ &= \bar{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^\pi(s') && (2.11) \end{aligned}$$

With the V and Q functions understood, we now inspect the Value Iteration (VI) algorithm. The Value Iteration algorithm initializes an estimation of $V(s)$ for each $s \in \mathcal{S}$ arbitrarily and repeatedly updates these estimates. This is done until some convergence criterion is met. Value Iteration itself returns only these estimations of $V(s)$ for all states, and not a policy. However, given these converged estimations, the optimal policy can be greedily extracted using equation 2.12. Pseudocode for the Value Iteration algorithm, adapted from Kaelbling et al.[18], can be found in Algorithm 1.

[1] All definitions in this subsection are adapted from Bellman[2] and Kaelbling et al.[18].

Algorithm 1 Value Iteration

```
1:  procedure VALUEITERATION(S, A, T, R, γ, ε)
2:      Initialize V(s) := 0 for all s ∈ S
3:      repeat
4:          V_prev(s) := V(s) for all s ∈ S
5:          for s ∈ S do
6:              for a ∈ A do
7:                  Q(s,a) := R̄(s,a) + γ ∑_{s'∈S} p(s'|s,a) V_prev(s')
8:              end for
9:              V(s) := max_a Q(s,a)
10:         end for
11:     until |V(s) − V_prev(s)| < ε for all s ∈ S
12:     return V
13: end procedure
```

$$\pi(s) = \underset{a}{\operatorname{argmax}} \left[\bar{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V(s') \right] \tag{2.12}$$

Similarly to VI, an algorithm named Policy Iteration also exists. This algorithm iteratively evaluates a policy and then updates the policy until it converges. Policy evaluation boils down to VI with a fixed policy; following a given π rather than taking the optimal action over the current Q estimates. This is referred to as simplified Value Iteration. Pseudocode for the Policy Iteration algorithm can be found in Algorithm 2. More details can be found in Chapter 17 of Russell and Norvig[31].

2.3 Probabilistic Programming

Occasionally, one could face a problem where one has some knowledge Y and wants to use this to infer some other knowledge X. For example, one could wonder what the probability is that a burglary happened at their home, given that their neighbor called about the alarm system sounding. Likewise, one could wonder what the probability is that it has rained, given that the grass is wet [30]. These kinds of problems can be answered by characterizing the posterior probability distribution $P(X|Y)$; the goal is to *infer* X, *conditioned* on Y.

Probabilistic programming is a useful paradigm for such probabilistic modeling and inference problems. At the heart of a probabilistic program lies a generative model. This is a function that can generate data from a distribution specified as a program. This means that the distribution can be specified using a combination of sample statements and deterministic statements. Sample statements define random variables. An assignment of values to the random variables is called an execution or a trace of the program. Next to the ability to sample from distributions, the other added construct that distinguishes probabilistic programming from 'regular' programming is the ability to condition on values of random variables. This

Algorithm 2 Policy Iteration

1: **procedure** POLICYITERATION($\mathcal{S}, \mathcal{A}, T, R, \gamma, \varepsilon$)
2: Initialize policy $\pi(s)$ arbitrarily for all $s \in \mathcal{S}$
3: **repeat**
4: **// Policy Evaluation (Simplified Value Iteration)**
5: Initialize $V(s) := 0$ for all $s \in \mathcal{S}$
6: **repeat**
7: $V_{prev}(s) := V(s)$ for all $s \in \mathcal{S}$
8: **for** $s \in \mathcal{S}$ **do**
9: $V(s) := R(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, \pi(s)) V_{prev}(s')$
10: **end for**
11: **until** $|V(s) - V_{prev}(s)| < \varepsilon$ for all $s \in \mathcal{S}$
12:
13: **// Policy Improvement**
14: $policy_is_unchanged := true$
15: **for** $s \in \mathcal{S}$ **do**
16: $a := \pi(s)$
17: $\pi(s) \leftarrow \text{argmax}_{a'} R(s, a') + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a') V(s')$
18: **if** $a \neq \pi(s)$ **then**
19: $policy_is_unchanged := false$
20: **end if**
21: **end for**
22: **until** $policy_is_unchanged$
23: **return** π, V
24: **end procedure**

is done using `observe` and `condition` statements[2]. `condition(b)` invalidates traces of the program in which the Boolean expression `b` is `false`. When a variable x is sampled from a continuous distribution and we want to condition x to be equal to some fixed value a, we could use `condition(`$x == a$`)`. However, we might want to allow some margin of error around a as well. In these cases, e.g. `observe(Normal(`x`, 0.1), `a`)` can be used; `observe` conditions a variable to follow a certain distribution. [15, 16]

To understand these concepts better, let us inspect an example. The example in Algorithm 3 is slightly adapted from Gordon et al. [16]. In this example, two values are sampled from a Bernoulli distribution. Next, the sum of these values is calculated and it is observed that the sum is always larger than 0. Finally, the sampled values are returned. An execution of this program will return `(1, 0)`, `(0, 1)`, or `(1, 1)` each with a probability of $\frac{1}{3}$. Note that a return value of `(0, 0)` has a probability of 0.0; this is ensured by the `condition` statement.

Now that we have an intuitive understanding, we proceed to a more formal characterization. A probabilistic program specifies a distribution over traces. Let X be the set of random

[2] `observe` and `condition` are sometimes used interchangeably in existing literature. E.g. Gordon et al.[16] call `observe` what Goodman et al.[15] call `condition`. We stick to the terminology from Goodman et al.[15].

Algorithm 3 Simple Probabilistic Program

1: **procedure** FLIP2COINS
2: $c_1 \sim Bernoulli(0.5)$
3: $c_2 \sim Bernoulli(0.5)$
4: $c_{sum} = c_1 + c_2$
5: **condition**($c_{sum} > 0$)
6: **return** c_1, c_2
7: **end procedure**

variables in a trace that are not observed, and Y be the set of observed random variables. A probabilistic program defines the joint distribution:

$$p(X,Y) = p(Y|X)p(X) \tag{2.13}$$

$$= \prod_{y\in Y} p(y|\mathtt{PA}(y)) \prod_{x\in X} p(x|\mathtt{PA}(x)) \tag{2.14}$$

where $\mathtt{PA}(x)$ is the set of random variables that precede and influence x.

Meanwhile, the overall goal of probabilistic programming is to perform probabilistic inference. For example, we might need the mean or mode of some variable under specified conditions. Or we may need to be able to draw samples from the posterior distribution. In other words, the goal is to characterize the posterior: [38]

$$p(X|Y) = \frac{p(X,Y)}{p(Y)}, \tag{2.15}$$

where

$$p(Y) = \int p(X,Y)dX. \tag{2.16}$$

Fortunately, many out-of-the-box inference procedures exist. For an overview of some well-known inference algorithms, we refer to chapter 8 of Goodman et al.[15]. The Metropolis-Hastings algorithm is highlighted in Section 2.4.

2.3.1 Factoring

Recall that `condition(b)` invalidates traces of the program in which the Boolean expression b is `false`. Occasionally, one might not want such hard constraints, and simply make traces where b is `false` less probable rather than invalid. In such cases, the `factor(·)` functionality can be used. The `factor(·)` functionality provides a 'soft' version of conditioning. Specifically, `factor(n)` adds n to the unnormalized log probability of the trace [14, 15]. Note that n is a numerical value and not a Boolean expression.

To understand how this functionality works, let us inspect an example (Algorithm 4). Ignoring the `factor(·)` statements, we can see that $p(c_1 = true) = p(c_1 = false) = 0.5$. Now we take the natural logarithms to find the log probabilities of these traces:

$$\ln(p(c_1 = true)) = \ln(0.5) \tag{2.17}$$

$$\ln(p(c_1 = false)) = \ln(0.5) \tag{2.18}$$

Algorithm 4 Simple Probabilistic Program that Uses Factor

```
1: procedure FLIPANDFACTOR
2:     c_1 ~ Bernoulli(0.5)
3:     if c_1 then
4:         factor(1.0)
5:     else
6:         factor(0.0)
7:     end if
8:     return c_1
9: end procedure
```

Next, we add the factors to the corresponding log probabilities. Since we are adding to the log probabilities, they become unnormalized log probabilities:

$$\ln(p_{unnorm}(c_1 = true)) = \ln(0.5) + 1 \tag{2.19}$$

$$\ln(p_{unnorm}(c_1 = false)) = \ln(0.5) + 0 \tag{2.20}$$

Getting rid of the logarithms on the left sides of the equations, we get:

$$p_{unnorm}(c_1 = true) = e^{\ln(0.5)+1} \tag{2.21}$$

$$= e^{\ln(0.5)+\ln(e^1)} \tag{2.22}$$

$$= e^{\ln(0.5e)} \tag{2.23}$$

$$= 0.5e \tag{2.24}$$

and

$$p_{unnorm}(c_1 = false) = e^{\ln(0.5)+0} \tag{2.25}$$

$$= 0.5 \tag{2.26}$$

Finally, normalizing these probabilities yields:

$$p(c_1 = true) = \frac{0.5e}{0.5e + 0.5} \tag{2.27}$$

$$= \frac{e}{e+1} \tag{2.28}$$

$$\approx 0,73 \tag{2.29}$$

and

$$p(c_1 = false) = \frac{0.5}{0.5e + 0.5} \tag{2.30}$$

$$= \frac{1}{e+1} \tag{2.31}$$

$$\approx 0,27 \tag{2.32}$$

As we can see, the trace where $c_1 = false$ is now successfully made less probable than the other trace. Nevertheless, it still has a non-zero probability.

2.4 Metropolis-Hastings

A popular method for sample-based approximate probabilistic inference is Markov Chain Monte Carlo (MCMC). On a high level, the goal is to find the right Markov Chain [29] such that the stationary distribution of this Markov Chain is equal to the target distribution: the distribution that we want to sample from. A recipe to find this Markov Chain is the Metropolis-Hastings (MH) algorithm [15]. The goal of the Metropolis-Hastings algorithm is to draw samples from a target distribution $p(x)$. Given an initial sample x, first, $x_{proposal}$ is sampled from $q(x_{proposal}|x)$, which is known as the proposal distribution. The proposal $x_{proposal}$ is accepted or rejected with probability $\alpha(x_{proposal}|x)$ which is calculated as: [31]

$$\alpha(x_{proposal}|x) = \min\left(1, \frac{p(x_{proposal})q(x|x_{proposal})}{p(x)q(x_{proposal}|x)}\right) \tag{2.33}$$

If $x_{proposal}$ is rejected, the sample that is drawn after x is again x. Meanwhile, if $x_{proposal}$ is accepted, the sample that is drawn after x is $x_{proposal}$. Then, the procedure is repeated in order to draw the next sample. If the sample x that was used to initialize this procedure has a very low probability under the target distribution $p(x)$, then the first samples from the MH algorithm should be discarded; the Markov Chain has to converge to its stationary distribution first. This is known as the burn-in phase. [15]

2.5 Bayesian Policy Search with Policy Priors

A fundamental building block of this thesis is the work of Wingate et al.[39], titled "Bayesian Policy Search with Policy Priors". Their work presents a technique for policy search with a prior distribution over policies.

A policy π is represented as a dictionary, mapping $\pi(s)$ to an action for each $s \in \mathcal{S}$. Each $\pi(s)$ of a policy is sampled from a $categorical(\theta)$ distribution. One could embed prior knowledge of the policy into the problem by biasing θ. For example, if it is known that an agent should generally move up in a grid-world, θ could be set to $[0.4, 0.2, 0.2, 0.2]$ over the ordered action set $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$. If we have no prior knowledge of the policy, θ would be set to $[0.25, 0.25, 0.25, 0.25]$. After a policy is sampled, it needs to be evaluated; the quality of a policy guides the search procedure. The work assumes that the value of a policy (equation 2.4) can be evaluated. In a practical reinforcement learning setting, this would be estimated using sampled trajectories of the policy.

So far, there is the assumption that a domain expert can define a good action distribution θ manually. To eliminate this assumption, the value of θ is *learned* by the algorithm. To this end, a hierarchical Bayesian model is used. As before, $\pi(s)|\theta$ follows a $categorical(\theta)$ distribution for each s. As an added layer above this, θ follows a $Dirichlet(\cdot)$[3] distribution. θ is now a random variable rather than a manually defined parameter. As a result, θ can be learned besides π. This learning is done using the Metropolis-Hastings algorithm where

[3]Think of the *Dirichlet* distribution as a probability distribution over vectors that sum to one. For more information, see Bishop[3].

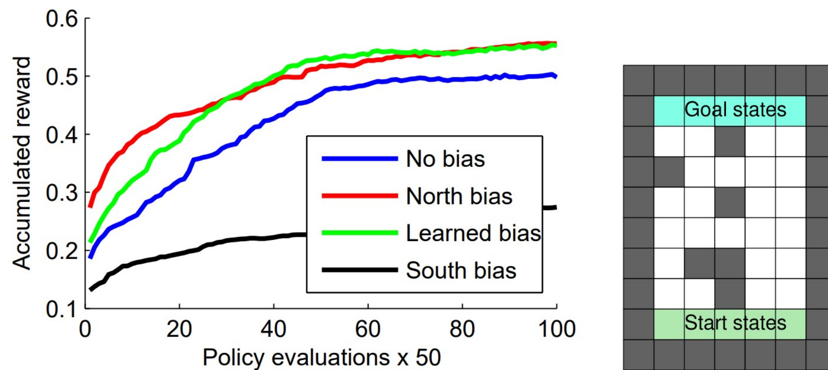

Figure 2.1: The MDP on the right is solved with four set-ups. The "No bias" case is the baseline (unbiased MH). The "North bias" and "South bias" cases bias the prior over policies to the North and South respectively. The "learned bias" case shows what happens when θ is learned besides π. Image copied from Wingate et al.[39].

updates are done both to π (to learn better policies) and to θ (to learn the prior action distribution). As mentioned before, the quality of the policy guides this search procedure.

For an interesting example see Figure 2.1. From this, it can be seen that a good manually defined bias (North bias) helps, but a bad one (South bias) deteriorates the performance. The learned bias asymptotically approaches the manually defined good bias. Intuitively what happens is that "learning that north in one part of a maze is good will bias the search towards north in other parts of the maze"[39].

The difference between the work of Wingate et al.[39] and this thesis is that Wingate et al. use a single θ for the whole state space. In contrast, this thesis allows a user to define n subsets/regions of the state space, with n θs; one θ per region. This does not lose generality, as it could still turn out that $\theta_i \approx \theta_j$ where $i \neq j$ after inference.

Chapter 3

Related Work

This chapter maps out related literature about Planning as Inference, Planning by Probabilistic Programming, Leveraging Prior Policy Knowledge, and State Abstraction for MDPs. These topics are discussed in Sections 3.1, 3.2, 3.3, and 3.4 respectively. Furthermore, the most notable differences between the related works and this thesis are explained.

3.1 Planning as Inference

Many methods exist that cast the problem of planning as a probabilistic inference problem. They all differ from this thesis in that the following methods do not make use of probabilistic programming (Section 2.3).

Botvinick and Toussaint[4] explain the concept of planning as inference as using a generative model to allow the agent to attach a probability score to any action-outcome-reward sequence. With this, one can *condition on reward* and infer the actions that lead to obtaining the reward. This can be mathematically formalized as minimizing Kullback-Leibler divergence between the marginal distribution over states and actions under the agent's current policy, and the corresponding posterior distribution conditioned on reward. They also show that approaching planning as a problem of probabilistic inference is more similar to how a brain performs planning. Finally, they argue that sample-based approximate inference might be related to how humans process information with limited rationality.

Attias[1] performs planning as inference by treating actions a_t as hidden variables and inferring them. Only discrete states and actions are considered, and planning is considered as reaching a goal state rather than finding a policy for an MDP that optimizes the expected return. Actions are inferred using Maximum-A-Posteriori, conditioned on the given initial state and the fact that the agent *should be* in the goal state at time step T. The limitation of this approach is that the time to reach the goal is typically unknown beforehand; then the algorithm would need to run multiple times, decreasing T at every iteration. Furthermore, the technique is different from planning in MDPs because it infers an action sequence and not a policy.

Comparably to Botvinick and Toussaint[4], the work of Toussaint and Storkey[35] also translates planning to probabilistic inference. Where Attias[1] inferred an action sequence

and total time T needed to be known beforehand, here a policy is inferred and T needs not to be known in advance. The approach uses an Expectation-Maximization algorithm, where in essence the E-step performs policy evaluation and the M-step performs a policy update. Under exact inference, this method is analogous to policy iteration. This analogy no longer holds under approximate inference. The most important contribution is that the use of approximate inference is now possible for planning in MDPs. The work of Toussaint et al.[36] generalizes this to POMDPs.

Levine[21] presented a technique for reinforcement learning as inference. His formulation of casting the policy search to probabilistic inference provides useful insights. An 'optimality' variable O_t is introduced, which denotes that the behavior of the agent at time step t is optimal. The distribution of this variable is defined as:

$$p(O_t = 1|s_t, a_t) = \exp(R(s_t, a_t)) \tag{3.1}$$

Then the posterior distribution of a trajectory conditioned on $O_t = 1$ for all $t = 1, .., H$ is:

$$p(\tau|O_{1:H}) \propto p(\tau, O_{1:H}) = \rho(s_1)\prod_{t=1}^{H} p(O_t = 1|s_t, a_t)p(s_{t+1}|s_t, a_t) \tag{3.2}$$

$$= \rho(s_1)\prod_{t=1}^{H} \exp(R(s_t, a_t))p(s_{t+1}|s_t, a_t) \tag{3.3}$$

$$= \left[\rho(s_1)\prod_{t=1}^{H} p(s_{t+1}|s_t, a_t)\right] \exp\left(\sum_{t=1}^{H} R(s_t, a_t)\right) \tag{3.4}$$

From this, we can see that the probability of observing trajectory τ given that we behave optimally is the probability of τ given the dynamics of the world multiplied by the exponential of the sum of rewards obtained over τ. This idea of an optimality variable is loosely related to the way conditioning on reward is done in this thesis using the `factor` functionality of probabilistic programs as explained in section 4.3.1.

3.2 Planning by Probabilistic Programming

Little work on solving MDP planning problems with probabilistic programming exists. Thon et al.[34] presented a short paper arguing in favor of using the probabilistic programming language ProbLog to naturally represent and solve planning problems. However, in their work, this again entails planning in the sense of inferring an action sequence rather than inferring a policy. Furthermore, there is no mention of MDPs, only planning to reach one of a set of goal states.

The work of Nitti et al.[26] introduces HYPE, which is an off-policy online technique for sample-based planning for MDPs using the probabilistic programming language ProbLog. An importance sampling strategy is used, where the probabilities of the transition model are exploited for the weights. This is in contrast to similar methods that only need to be able to sample from the transition model instead of accessing the probabilities, such

as Sparse Sampling, as introduced by Kearns et al.[19]. The virtue of HYPE is that it allows estimating Q–values for states that have never been accessed before. It works well in discrete, continuous, and hybrid domains.

Similar to but different from HYPE is the work of Bueno et al.[7]. It extends the probabilistic programming language ProbLog to allow the representation of infinite-horizon factored MDPs. A method for VI is presented that makes use of weighted model counting. Although only a simple VI scheme is presented, the mapping to this framework opens the door to more sophisticated inference approaches. In comparison, HYPE[26] is for domains involving mixtures of discrete and continuous state variables whereas the work of Bueno et al.[7] is restricted to boolean factored state representations. HYPE uses importance sampling and Monte-Carlo methods to solve finite-horizon problems whereas MDP-ProbLog uses weighted model counting techniques to solve infinite-horizon problems.

The work of van de Meent et al.[37] introduced Black Box Policy Learning/Black Box Policy Search, which uses black-box variational inference in a probabilistic program to infer a policy. Both the world (i.e. the dynamics of the MDP) and the agent are modeled as probabilistic programs. The probabilistic program representing the agent is parameterized, while the distribution of this parameter is variationally learned.

3.3 Leveraging Prior Policy Knowledge

Rodriguez-Sanchez et al.[27] presented RLang, which is a framework for incorporating many variants of (partial) domain knowledge into reinforcement learning tasks. Many of the domain knowledge categories encompass knowledge of the environment, which is irrelevant in planning problems since the full dynamics of the environment are known. However, one category is prior policy knowledge. This is incorporated by manually defining an initial advice policy. This advice policy is then probabilistically mixed with a learned policy network using a mixing parameter that is annealed over time.

Similarly, Yang et al.[40] proposed Policy-Guided Planning for Generalized Policy Generation (PG3). This technique performs generalized policy search where a candidate policy needs to be provided and guides the search procedure. The score of a policy is then based both on its quality (from policy evaluation) and on its similarity with the candidate policy. This technique can be useful when there is already a high-level idea of the form of the policy, but some gaps still need to be filled in. Unfortunately, this method is designed only for PDDL[24] domains and not for MDPs. This means that stochasticity in the environment is unsupported. The most prominent difference between PG3 and this thesis is that this thesis considers MDPs, and hence *does* support stochastic environments. Furthermore, PG3 is a method for *generalized planning*, which means that multiple problems from the same family of planning problems are given to the algorithm and the solution is a *generalized policy* that works well for all problems. Meanwhile, the techniques presented in this thesis can be seen as classical policy search with elements that can be useful for generalization.

Finally, rather than leveraging an advice/candidate policy, Wingate et al.[39] presented a technique for policy search with a prior distribution over policies. This technique is explained in detail in Section 2.5.

3.4 State Abstraction for MDPs

According to Giunchiglia and Walsh[13] "one can think of abstraction as the process which allows people to consider what is relevant and to forget a lot of irrelevant details which would get in the way of what they are trying to do." In the context of MDPs, we can think of abstraction as simplifying an MDP while retaining (to some extent) the essential information. Having a simplified version of an MDP can result in more efficient solving and in a more concise and interpretable policy. Meanwhile, too much simplification can result in the loss of essential information and in suboptimal behavior. This loss of essential information is known as *hidden state* as explained and examined by McCallum[23].

A well-studied type of MDP abstraction is state abstraction. In this case, *ground states* are clustered together to form *abstract states*. The ground states are the states of the original MDP and the abstract states are the states of the simplified MDP [22, 8]. Interesting overviews of state abstraction approaches for MDPs are provided by Li et al.[22] and by Congeduti and Oliehoek[8]. Moreover, the work of Starre et al.[32] shows how state abstractions can efficiently and effectively be applied to model-based reinforcement learning.

One method to discover state abstractions is proposed by Jong and Stone[17]. By eliminating state factors of a factored MDP that prove to be irrelevant, the MDP is simplified. Irrelevance of state factors is intuitively defined as a state factor for which an agent can still behave optimally if it ignores it. This irrelevance is related to the `UNLOCK` and `LAVA` MDPs (Section 5.1) evaluated in this work, where respectively the state factors *has_key* and *y* are irrelevant.

This thesis is different from state abstractions in that the ground MDP is still solved, rather than a simplified MDP. Nevertheless, the division of the state space into regions where actions are similar, as proposed in this thesis, can be interpreted as a layer of abstraction.

Chapter 4

Methodology

This chapter explains how policy search for MDPs is done with probabilistic programs. The two main ingredients for probabilistic programs are the generative model and the inference procedure. Section 4.1 introduces the problem setting and specifies performance metrics for evaluation, and Section 4.2 explains the overall approach. Then, Section 4.3 states the generative model and explains it in detail. Additionally, Section 4.4 explains the inference procedure that is used to infer good policies, given the soft prior knowledge. Finally, Section 4.5 explains how the soft prior knowledge itself can be (partially) conditioned and inferred. For modeling and inference, the Probabilistic Programming Language Gen [9] is used.

4.1 Problem Setting

This thesis considers the problem class of finding policies for factored Markov Decision Processes (Section 2.1) with and without a given transition model of the environment. Within this problem class, we can define two cases specifically: (1) factored MDPs where a user has some soft prior knowledge (Section 4.2.2 specifies how this is represented) and (2) factored MDPs where one can benefit from uncovering such knowledge for a better insight into the underlying problem or for generalization purposes.

For the first case, we are interested in the acceleration of the policy inference. This is measured by comparing the convergence progress of the policy inference that leverages the prior knowledge of the policy, to an analogous algorithm where the prior knowledge of the policy is not considered. Moreover, this is evaluated in terms of wall-clock time and in terms of the number of policy evaluations.

For the second case, we are interested in the correctness of the inferred knowledge. This is simply measured as the amount of times the inferred knowledge is correct, out of the total amount of repetitions of the algorithm.

4.2 The Approach: Priors over Regions

The approach by Wingate et al.[39] as explained in Section 2.5 introduces a prior parameter θ representing the action distribution. The inference is done on the action distribution θ

as well as the policy π by means of the Metropolis-Hastings algorithm (Section 2.4). The result of this is that after learning that action a performs well in one part of the state space, the search is biased towards a in other parts of the state space as well. This formalization implicitly assumes that we can expect generalization of good and bad actions over the entire state space. This might not always be the case.

Rather than considering the entire state space, we could also consider subsets of $\mathcal{S}$. If we can identify regions of the state space in which we expect actions to be similar, we can associate each region i with its own action distribution parameter vector θ_i. This is a clear extension of the approach of Wingate et al.[39]. The divisions of the state spaces into regions are a form of soft prior knowledge of the policy; it formulates a suspicion that a certain subset of $\mathcal{S}$ has some similarity towards the goal. Furthermore, if it turns out that the division of regions is unjustified, it can still be inferred that $\theta_i \approx \theta_j$ where $i \neq j$. Hence, we do not expect a lot of deterioration when the division is wrong.

Another beneficial property of this approach is that it is asymptotically correct. In this case, it means that if we run the algorithm long enough, we are guaranteed to find the optimal policy eventually. This property is a direct result of following the Metropolis-Hastings algorithm.

4.2.1 Motivating Examples

To illustrate this idea of dividing a state space into regions, let us have a look at a simple example (Figure 4.1 (left)). In this example, the states are the white grid cells and the actions are $\{\uparrow, \downarrow, \leftarrow, \rightarrow, IDLE\}$ and there is a treasure in the top-right corner. This is where the agent receives a large reward. Furthermore, the transitions are stochastic; there is a 20% chance an agent will "slip" and move one step in any perpendicular direction. Within a world like this, we do not see a clear dominant action a, like the action $\uparrow$ in the example from Figure 1.1 (left). The only clear aspect we can infer about the action distribution if we consider it over the entire state space is that *IDLE* is never a good action. However, perhaps we can come up with different strategies to divide the state space into regions in which we might expect some more generalization of good and bad actions. These ways to divide the state space we name "region configurations" (RC). A naive region configuration would be to consider the entire state space as one big region[1]. If we divide the state space according to one of the RCs as marked in Figure 4.1 (right), we can expect more generalization of good actions *within* the regions. For example, let us consider region 1 in RC 3 (the yellow strip). When it is found out that $\rightarrow$ is a good action in the right part of this strip, it makes sense to bias the action distribution over this region in favor of $\rightarrow$, so that it becomes more likely in the left part of the yellow strip as well. Similar reasoning holds for the other regions.

As a second example, imagine a grid-world with a key in the top-left corner and a treasure in the bottom-right corner (Figure 4.2). The state space of this MDP is the Cartesian product of the grid-cells and $\{has_key, !has_key\}$. The possible actions are $\{\uparrow, \downarrow, \leftarrow, \rightarrow, PICKUP, OPEN, IDLE\}$, where *PICKUP* picks up the key when the agent is on top of it and is identical to *IDLE* in all other cases. Furthermore, *OPEN* yields a

[1] We refer to this naive region configuration as RC 1, it functions as a baseline.

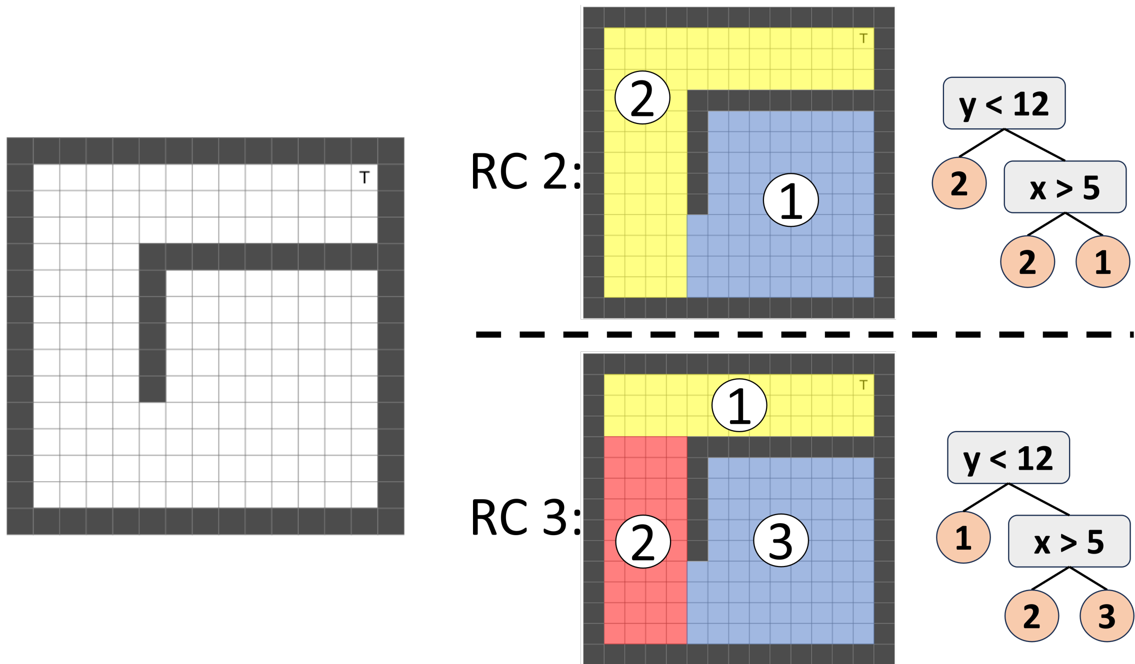

Figure 4.1: *Left:* an example grid-world with a treasure in the top-right corner. *Right:* two example ways to divide the state space into regions; visualized in the grid-world and represented as a decision tree over state factors. For ease of reference, regions are labeled with numbers.

large reward when the agent is on the treasure and has the key, or when the agent is on the treasure and it is unlocked. Similarly, *OPEN* is identical to *IDLE* in all other cases. The stochasticity of the state transitions is identical to that of the previous MDP. Again, except for *IDLE*, there are no clearly good or bad actions that can be inferred if we consider the state space in its entirety; when we do not have the key, the agent should clearly favor $\uparrow$ and $\leftarrow$ and when we have the key, the agent should clearly favor $\downarrow$ and $\rightarrow$. Making a split based on the state-factor *has_key*, therefore, seems a sensible choice. Furthermore, we could split the state space even further by taking into account that the locations of the key and treasure are special states. These two region configurations are visualized in Figure 4.3.

4.2.2 Representation of the Soft Prior Knowledge

The division of $\mathcal{S}$ into regions is visually represented for two example MDPs in Figure 4.1 (right) and Figure 4.3. The next step is a method to represent this soft prior knowledge more formally.

The approach used in this thesis is to represent it as a decision tree [6] where a decision node is based on a factor of the MDP and a leaf node is a region label. Region label i means that the region is associated with the action distribution parameter vector θ_i. The region configurations visualized in Figures 4.1 (right) and 4.3 are accompanied by their

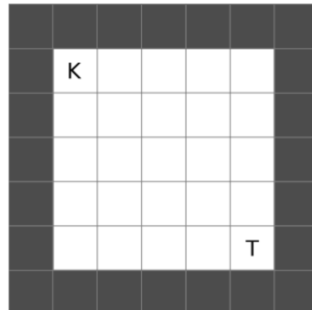

Figure 4.2: An example MDP with a key in the top-left corner and a treasure in the bottom-right corner. Note that this grid visualizes the *x* and *y* state-factors and not the *has_key* state factor.

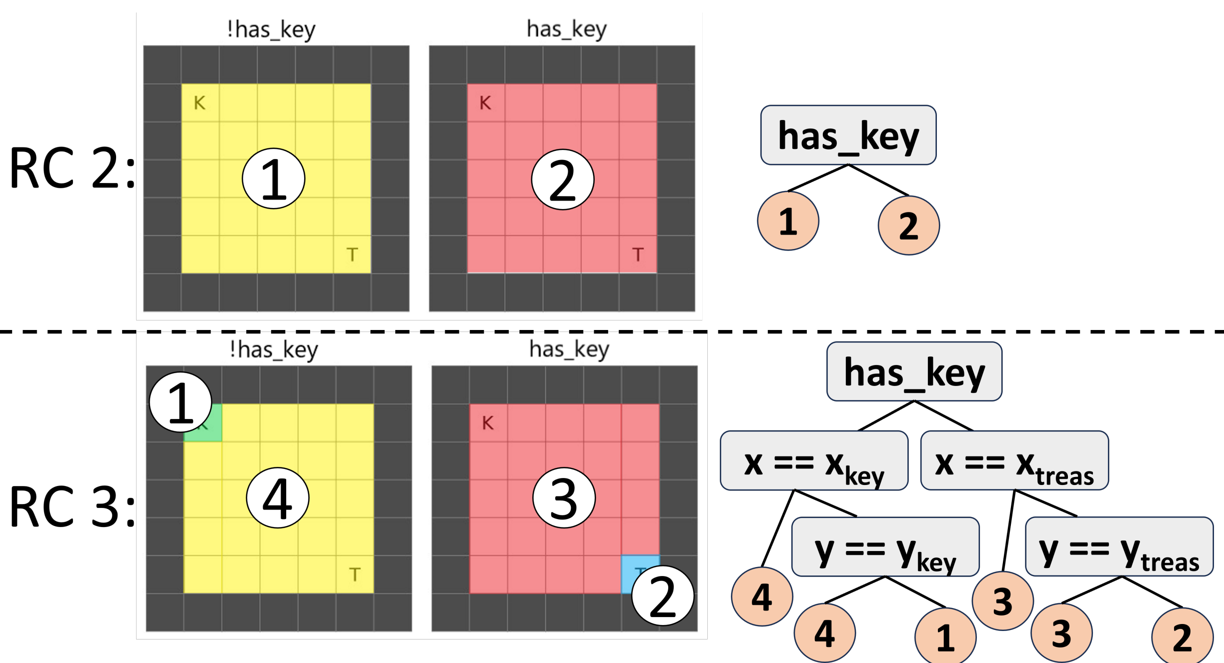

Figure 4.3: Two examples of ways to divide the state space from Figure 4.2 into regions; visualized in the grid-world and represented as a decision tree over state factors. For ease of reference, regions are labeled with numbers.

corresponding decision trees over state factors[2]. The trees in the figures follow the standard convention; the left of a decision node means *false* and the right of a decision node means *true*.

Besides the visual representation of the tree, they can also be represented formally. In this thesis, any tree is of an abstract type `Node` which has two concrete subtypes: `LeafNode` and `DecisionNode`. A `LeafNode` has only one attribute: `region_index`, i.e. a label of the region. Meanwhile, a `DecisionNode` has five attributes:

- `state_factor`, which stores a state factor of the MDP (Section 2.1);
- `pivot_value`, which stores a value from the domain of `state_factor`;
- `comparator`, which stores a function that can compare two values (for now: $<$, $>$ or $==$);
- `left`, which stores a `Node` representing the left sub-tree;
- `right`, which stores a `Node` representing the right sub-tree.

Since `left` and `right` are of type `Node`, they could be leaves or decision nodes. Hence, this is a recursive data structure.

4.3 Generative Model

The generative model for the policy search (Algorithm 5) first samples a decision tree that divides the state space into regions. Then, for each region i, an action distribution parameter vector θ_i is sampled uniformly from the $(|\mathcal{A}|-1)$-simplex. With these prior action distributions per region, a policy π is sampled. Finally, the policy π is evaluated and the result of the evaluation procedure is used to make better policies more likely. The *evaluate_policy* function is now a simplified value iteration procedure where the policy is fixed to the given π (Section 2.2). In situations when the transition dynamics are unknown but an agent can interact with the environment, this policy-evaluation step could also be based on sampled trajectories. Firstly, Section 4.3.1 explains how the `factor` functionality is used to make better policies more likely. Secondly, Section 4.3.2 explains why and how the decision trees are sampled.

4.3.1 Making Better Policies More Likely

As the goal is to generate good policies, we need to condition the distribution on the policy being good; we want to maximize the policy quality. This conditioning can be done in various ways. Perhaps the most naive way is to sample a start state s_0 from ρ (Section 2.1), and condition that $V^{\pi}(s_0)$ is equal to some high value. Somewhat defeating the purpose, let us assume that the best possible value $V^{\pi^*}(s_0)$ is already known beforehand[3]. Given this value,

[2] Note that *has_key* in the decision trees in Figure 4.3 is syntactic sugar for *has_key* $== 1$

[3] This is typically not the case as you would need to know the optimal policy, which is what we are trying to find in the first place.

Algorithm 5 Generative Model for Good Policies

```
 1: procedure GENERATE_POLICY(mdp, υ, max_num_regions, max_tree_depth)
 2:     Sample tree ~ generate_tree(mdp, max_num_regions, max_tree_depth)
 3:
 4:     // Sample θs
 5:     Let α be a vector of ones of length |A|
 6:     for region_index ∈ {1..max_num_regions} do
 7:         Sample θ[region_index] ~ Dirichlet(α)
 8:     end for
 9:
10:     // Sample policy
11:     for s ∈ S do
12:         Sample π(s) ~ categorical(θ[tree(s)])
13:     end for
14:
15:     // Make better policies more likely
16:     Let V^π = evaluate_policy(π)
17:     for s ∈ S do
18:         factor(υ · V^π(s))
19:     end for
20:
21:     return π, V^π
22: end procedure
```

it can be conditioned that $V^{\pi}(s_0)$ is equal to $V^{\pi^*}(s_0)$. Rather than a `condition`$(V^{\pi}(s_0) == V^{\pi^*}(s_0))$ statement, one should use an `observe`$(N(V^{\pi}(s_0), \sigma^2), V^{\pi^*}(s_0))$ statement with some large σ^2. This provides "a hill to climb" for the algorithm, rather than providing an equality that should magically hold (Section 2.3). More realistically, if $V^{\pi^*}(s_0)$ is unknown beforehand, an estimate of $V^{\pi^*}(s_0)$ (that needs to be at least somewhere in its neighborhood) could also be used analogously. Multiple values can be manually tried or a line search can be done. All of these methods are somewhat of a workaround, and not a natural solution to the underlying maximization problem.

Since we are facing a maximization problem, conditioning on a variable assuming some given value is not the most natural way. More natural would be to use the `factor` functionality. Recall that `factor`(n) adds n to the unnormalized log probability of the trace [14, 15] (Section 2.3.1). This functionality makes it much more natural to phrase the maximization of expected return as an inference problem because no assumptions about the knowledge of values need to be made. In this thesis, `factor`$(\upsilon \cdot V^{\pi}(s))$ for all $s \in \mathcal{S}$ is used, where υ is the rationality parameter. Firstly, the rationality parameter υ is introduced to strike a balance between randomness ($\upsilon = 0$) and perfect maximization ($\upsilon = \infty$) [11]. Secondly, the factoring is done for all $s \in \mathcal{S}$ rather than only for s_0 because policies that have high values for many states but not for s_0 are still preferred over policies that have low values for all states.

Since the probabilistic programming language Gen does not have built-in factor functionality, the same functionality is achieved indirectly. This is inspired by van de Meent et al.[38]. First, let us recall that the probability density function of an $exponential(\lambda)$ distribution is:[28]

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x}, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{4.1}$$

Hence, conditioning a random variable that is sampled from this distribution to be equal to zero yields $\lambda e^{-\lambda \cdot 0} = \lambda$. Thus, for any trace in which a random variable is sampled from a $exponential(\lambda)$ distribution and conditioned to be equal to 0, the unnormalized probability of the trace will get multiplied by λ. Equivalently, we can say that $\ln(\lambda)$ is added to the unnormalized log probability of the trace. This is leveraged to achieve the factor functionality by sampling an auxiliary random variable $factor_s$ from an exponential distribution with a λ-parameter of $exp(\upsilon \cdot V^{\pi}(s))$. By conditioning that $factor_s$ is equal to 0, the unnormalized probability of the trace then gets multiplied with $exp(\upsilon \cdot V^{\pi}(s))$. Analogously, we can say that $\upsilon \cdot V^{\pi}(s)$ is added to the unnormalized log probability of the trace. This is exactly the desired behaviour of `factor`$(\upsilon \cdot V^{\pi}(s))$ (Section 2.3.1). By repeating this for all $s \in \mathcal{S}$ (with $\upsilon > 0$) and observing all these factors to be equal to 0, we have successfully made better policies more likely.

4.3.2 Sampling the Trees

The decision tree representing the soft prior knowledge decides which states are "batched together" in regions; it decides which θ_i corresponds to a state s. Examples of such trees are shown in Figure 4.1 (right) and Figure 4.3. An expert in the domain of the MDP might be able to define such a tree. When such a tree is given, we can simply condition the sampled tree in the probabilistic program to be equal to the given tree. Next, we can perform policy inference that leverages this provided soft prior knowledge (Section 4.4).

In situations when it is not the case that a (complete) tree is given, inferring these trees could be an interesting research direction since such an abstract representation of the problem can provide insight into the behavior of the agent (Section 4.5). To open the door to this kind of inference, the trees need to be a part of the trace of the probabilistic program. When a tree is given by a domain expert, this information can still be used by conditioning the program on the given tree. One could even imagine constructions where a partial tree is given and the rest is inferred (Section 4.5.1).

To make the tree part of the trace, a generative model for decision trees is needed (Algorithm 6). In essence, it is first decided if the current node should be a leaf node or a decision node. If it is a leaf node, the index of the region is sampled. If it is a decision node, then a state factor is sampled, along with a pivot value from the domain of the sampled state factor. Then a comparator (for now: $<$, $>$ or $==$) is sampled. Since the decision tree is a recursive data structure, the left and right sub-trees are then sampled with a recursive call to the generative model.

To make sure the trees do not grow infinitely deep, a max_tree_depth parameter is used. Furthermore, to encode a preference for shallow trees over deep trees, is_leaf is sampled

Algorithm 6 Generative Model for Region Configuration Trees on Factored MDPs

```
 1: procedure GENERATE_TREE(mdp, max_num_regions, max_tree_depth)
 2:     if max_tree_depth≤ 0 then
 3:         Sample is_leaf ∼ Bernoulli(1.0)
 4:     else
 5:         Sample is_leaf ∼ Bernoulli(0.6)
 6:     end if
 7:
 8:     if is_leaf then
 9:         Sample region_index ∼ uniform_discrete(1, max_num_regions)
10:         return LeafNode(region_index)
11:     else
12:         Sample a state_factor
13:         Sample a pivot_value (from the domain of state_factor)
14:         Sample a comparator
15:
16:         Sample left ∼ Generate_Tree(max_num_regions, max_tree_depth − 1)
17:         Sample right ∼ Generate_Tree(max_num_regions, max_tree_depth − 1)
18:
19:         return DecisionNode(state_factor, pivot_value, comparator, left, right)
20:     end if
21: end procedure
```

from a Bernoulli distribution that is biased towards *true*. This is done to prevent trees from ‘overfitting’. The Bernoulli parameter is set to 0.6. This value is chosen somewhat arbitrarily, but still to indicate a preference for shallow trees over deep trees. A better approach might be to refactor this as a hyperparameter. However, for simplicity, this is left as future work.

Note that the generative model can sample invalid trees. For example, a decision node `"x > 5"` can be sampled, followed by a decision node `"x > 6"` in the left subtree. This means that no states will ever reach the right sub-tree of `"x > 6"`. This limitation is however easily solved with a post-processing step.

4.4 Inferring Policies

Assuming that a decision tree encoding of the soft prior information is given, we can then condition Algorithm 5 such that the variable at address $tree$ is equal to the given tree. Given a region configuration (represented as a tree), this section explains how a good policy π and the prior parameter vectors θs are inferred.

4.4.1 Metropolis-Hastings on π and θ

First, a trace is generated where the tree is conditioned to be equal to the given tree and all factors are conditioned to be equal to zero. Furthermore, all θs are conditioned to be equal to vectors of length $|\mathcal{A}|$ repeating the scalar $\frac{1}{|\mathcal{A}|}$. In other words, all θs are initialized in the middle of the corresponding $(|\mathcal{A}|-1)$-simplex.

Second, a loop is initiated where this trace is updated iteratively. The number of times $evaluate_policy(\cdot)$ is called is counted; the loop terminates when this counter exceeds a given threshold or when a time limit is exceeded. The Metropolis-Hastings algorithm can be interpreted as a stochastic search for a good policy. Therefore, the final policy is not necessarily the best policy found overall. Hence, the best policy found so far is memorized during the search.

An update to the trace within this loop can be an update to the policy or an update to any of the θs. This is decided randomly. The probability of updating one or the other is set with a parameter (Section 4.4.3). This results in two possible update steps:

(1) When the policy needs to be updated, a state s is sampled first. Next, using the tree, the region i corresponding to s is retrieved. Then $\pi_{proposal}(s)$ is sampled from $categorical(\theta_i)$. Furthermore, $\pi_{proposal}(s') := \pi(s')$ for all $s \neq s'$. The proposed policy is accepted or rejected according to the Metropolis-Hastings algorithm.

(2) When the prior needs to be updated, a region index i is sampled first. Then $\theta_{i,proposal}$ is proposed as a random walk on the simplex starting at θ_i (Section 4.4.2). Again, the proposed vector is accepted or rejected according to the Metropolis-Hastings algorithm.

4.4.2 Random Walk over Simplex

To be able to perform random-walk Metropolis-Hastings steps on the θs, a random walk in the space of vectors summing to one would be needed. This means that given θ we need to propose a $\theta_{proposal}$ that lies in the neighborhood of θ. We assume the strategy proposed by Larget and Simon[20]. The proposal $\theta_{proposal}$ is sampled from a Dirichlet distribution where the concentration parameter is θ multiplied by a precision parameter:

$$\alpha = precision \cdot \theta \tag{4.2}$$

$$\theta_{proposal} \sim Dirichlet(\alpha) \tag{4.3}$$

Note that after defining α a check needs to be performed to make sure that no entry α_i is equal to 0.0. If that is the case, it is overridden to a very small value (1×10^{-7}) and α is re-normalized. Given this formulation, $\mathbb{E}[\theta_{proposal}] = \theta$ and $Var[\theta_{proposal}]$ scales approximately as $\frac{1}{precision}$ [12]. In this thesis, the precision parameter is always 30.0.

According to Fernandes and Atchley[12], this sampling strategy becomes inefficient as the dimensionality of the simplex increases. They also propose a different strategy for the random walk. For this thesis, the dimensionality of the simplex is always $|\mathcal{A}|$, and no experiments are done with large action spaces. Experimenting with the strategy from Fernandes and Atchley[12] and with larger action spaces, as well as optimizing the precision parameter are left as future work.

4.4.3 Implementation Details

The inference algorithm for inferring policies is explained in the previous sections. However, some implementation details are omitted because they are unimportant for a solid understanding of the overall approach. For completeness, these details are explained here.

Re-use of V Sampled policies are evaluated by means of simplified value iteration (Section 2.2). After the policy in the first trace is sampled, this policy evaluation procedure is initialized with $V^{\pi}(s) = 0.0$ for each state s. Afterward, every time the policy is updated, the policy evaluation procedure for $\pi_{proposal}$ is initialized with V^{π}; i.e. the values of the previous policy. When $\pi_{proposal}$ is accepted, V^{π} is overwritten to $V^{\pi_{proposal}}$ and memorized for the next update. This is a reasonable initialization because π and $\pi_{proposal}$ differ at most one entry of the dictionary. Consequently, it is likely that their values will often be similar[4].

Omitting Policy Re-evaluations When one of the θs is updated, this means that the policy is unchanged. In these cases, there is no need to re-evaluate the policy. However, when a trace is updated in Gen it re-runs the program and copies the values that are not updated from the previous trace. Unfortunately, this means that the policy evaluation procedure is re-run for every update, even when it is unnecessary as the policy is unchanged. To be able to configure when this procedure is run and when it is not, the values V need to be a part of the trace rather than the output of a deterministic procedure. To overcome this, the policy evaluation function is "disguised" as a probability distribution. A probability distribution *policy_evaluation_distribution* is defined. The distribution has no gradients and the logarithm of the probability density function is always 0.0. This ensures that this distribution does not influence the probabilities of traces. Furthermore, sampling from *policy_evaluation_distribution* given π and V_{init} evaluates the policy as expected and returns V^{π}. This formulation makes it easily configurable when and when not to execute the policy evaluation procedure; V^{π} can be "re-sampled" or copied from the previous trace, just like any other random variable in the trace.

Updating θ or π To decide whether θ or π needs to be updated, a hyper-parameter is introduced. This parameter defines the expected number of steps on the θs per $|\mathcal{S}|$ steps on π. We denote this parameter by ψ. The probability of updating π is then sampled from a $Bernoulli\left(\frac{|\mathcal{S}|}{|\mathcal{S}|+\psi}\right)$ distribution. This formulation is deemed more informative than sampling from e.g. $Bernoulli(\psi)$ directly because the size of the state space $|\mathcal{S}|$ is irrelevant in the current formulation. We also call this parameter ψ the inference balance parameter as it indicates how much computational effort is put into inferring the priors compared to how much computational effort is put into inferring the policy.

Picking $\pi(s)$ to Update When it is decided that an update to the policy needs to happen, the next question is for which s to update $\pi(s)$. The most simple solution would be to sample

[4]Note that this is not always the case. Occasionally, a small change can make a big difference. In these cases, the output of the policy evaluation procedure is still valid; it will simply take a bit longer to converge.

any state s uniformly from $\mathcal{S}$. A somewhat more sophisticated solution, which is used in this thesis, first samples a region uniformly. Next, a state s is sampled uniformly from all the states within this region. This strategy is used because it re-weighs the uniform distribution over $\mathcal{S}$ by taking into account the size of the regions.

Domain of υ The rationality parameter υ is explained to balance randomness ($\upsilon = 0$) and perfect maximization ($\upsilon = \infty$) (Section 4.3.1). Since factoring is implemented by sampling from an exponential distribution with a rate parameter of $exp(\upsilon \cdot V^{\pi}(s))$, a maximum value for υ needs to be considered. The maximum value of a `Float64` in Julia is 1.798×10^{308}. Now we solve the following equation for υ:

$$\exp(\upsilon \cdot V^{\pi}(s)) = 1.798 \times 10^{308} \tag{4.4}$$

$$\upsilon = \frac{\ln(1.798 \times 10^{308})}{V^{\pi}(s)} \tag{4.5}$$

When we take for example $V^{\pi}(s) = 100$, we get a maximum value for υ of 7.1. Hence, taking the rationality to large values is infeasible. Fortunately, the experiments in Chapter 5 show that $\upsilon = 1.0$ can already work very well.

4.5 Inferring Region Configurations

4.5.1 Conditioning Partial Trees

Since the decision tree representations of the soft prior knowledge are a part of the trace of the probabilistic program, we can perform conditioning and inference over the trees. The two simplest settings for conditioning are to have a given complete tree (as assumed in Section 4.4), or no given tree at all. In between these extremes is partial knowledge of the tree. In this section, various conditioning methods are illustrated.

Hard Conditioning Looking back at Algorithm 6, each of the variables that are defined by a sample statement can be fixed through conditioning. For example, `condition(is_leaf == 0)`[5] would make sure that the first node is a decision node and not a leaf. Likewise, the condition statement:

```
condition(is_leaf == 0, state_factor == x, pivot_value == 3,
    comparator == "==")
```

would define a root note `"x == 3"`, while leaving the rest of the tree up to inference. By extension, we can also fix the left sub-tree to be a leaf node immediately:

[5] Note that conditioning in Gen looks somewhat differently, in this case: `tree_constraints = Gen.choicemap(); tree_constraints[:is_leaf] = 0`. Notation is kept consistent with the prevailing notation as introduced in Section 2.3.

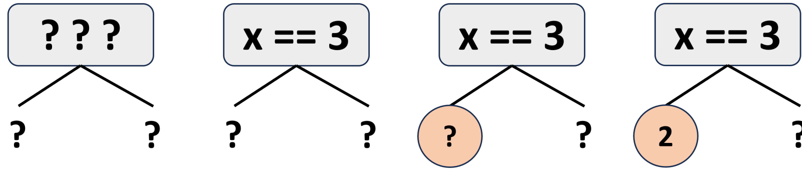

Figure 4.4: Four examples of partially conditioned trees. A question mark means that the value(s) under the address is unconditioned. An orange circle means that $is_leaf == 1$ at that address. Only a question mark and no circle means that is_leaf is unconditioned at that address and it can therefore become any sub-tree or a leaf.

```
condition(is_leaf == 0, state_factor == x, pivot_value == 3,
    comparator == "==", left=>is_leaf == 1)
```

Furthermore, we can even define the region index for the leaf node:

```
condition(is_leaf == 0, state_factor == x, pivot_value == 3,
    comparator == "==", left=>is_leaf == 1, left=>region_index == 2)
```

The aforementioned four examples are visualized in Figure 4.4 from left to right respectively. To avoid large written condition statements, six more examples of the possibilities with partial conditioning are visualized in Figure 4.5.

Soft Conditioning It is also possible that there is some knowledge about the tree that does not directly translate to the fixing of variables through condition statements. For example, we might have a suspicion that the tree should make a split on the state factor x somewhere, but we are not 100% sure and we do not know where in the tree this split should be located. Perhaps splitting on x only makes sense after we have split on *has_key*. Then, imposing a split on x in the root node would be undesired. In these cases, the `factor` functionality can be used again. After a tree is sampled, a tree traversal combined with if-statements can be used to decide if the tree matches our suspicions. If it does, `factor`$(\cdot)$ with some positive number between the parentheses will make those traces more important.

Similarly, we could have a suspicion for the value of a pivot without absolute certainty. For example, imagine a state factor x with a domain of $[1,5]$ and we suspect that a split should probably happen at $x == 3$. By default, the pivot for a split on x will be sampled from a $discrete_uniform(1,5)$ distribution. However, we can edit the generative function to sample the pivot from a $categorical([0.05, 0.2, 0.5, 0.2, 0.05])$ distribution if the sampled state factor is x. Nevertheless, updating the generative model in this way could be tedious. Instead, another strategy using `factor`$(\cdot)$ can be used again. After a tree is sampled, it can again be traversed. When a node with $state_factor == x$ is found, the probability of the corresponding pivot under the suspected distribution can be added to the unnormalized log probability of the trace. This probability can also be multiplied with some rationality factor, as was explained in Section 4.3.1.

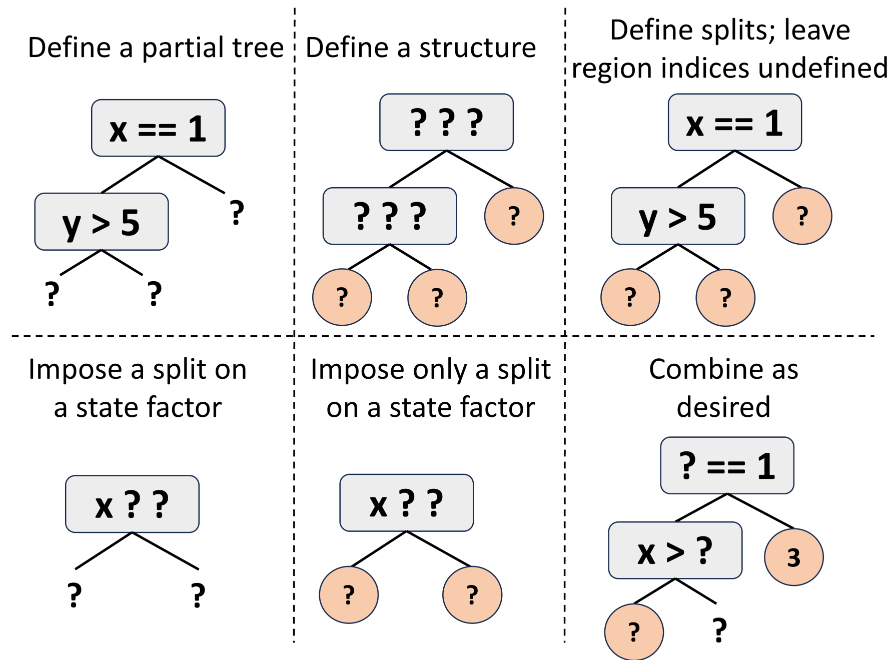

Figure 4.5: Six examples of partially conditioned trees. A question mark means that the value(s) under the address is unconditioned. Notation is identical to that of Figure 4.4.

4.5.2 Inference

Perhaps the most challenging aspect of inferring a tree is the stochastic support; the very existence of variables is uncertain. For example, imagine a decision tree in which the root is already a leaf node. There can be uncertainty about the *region_index* of the leaf, but at least it is known that a value needs to be filled in for the random variable *region_index*. Meanwhile, a variable at address *state_factor* does not even exist. The uncertainty about the very existence of random variables is something that can make inference challenging. We resolve this with two different approaches: circumventing stochastic support and ancestral sampling. These two approaches are explained respectively.

Circumventing Stochastic Support The first strategy is to circumvent the stochastic support by conditioning the structure of a tree (Figure 4.5). This way, the addresses of the random variables that are to be inferred are known beforehand. Inference on these random variables is again performed by means of Gibbs sampling.

Ancestral Sampling The second strategy is to again use the Metropolis-Hastings algorithm and use ancestral sampling for tree proposals. This means that trees are proposed by simply running Algorithm 6 forward, and not by proposing only a new value for a certain random variable. Hence, any proposed tree is independent of its predecessor. This method is not expected to scale well to large trees, as sampling the correct tree will become more

unlikely the larger the tree. Resolving this by means of a random walk in the space of decision trees is left as future work.

Finally, one could wonder how the computational effort is divided over inferring the tree, inferring the θs, and inferring π. Remember that previously (Section 4.4) the goal was to infer the θs while inferring π. To strike a balance on how much computational effort is put into inferring the θs or π, the inference balance parameter ψ was introduced (Section 4.4.3). Recall that ψ is the expected number of Metropolis-Hastings steps on the θs per $|\mathcal{S}|$ steps on π. Now, the goal of the algorithm is to infer the region configuration (represented as a decision tree) while inferring the θs while inferring π. Instead of introducing another hyperparameter to control this balance, the updates on the decision tree simply piggyback on the updates to the θs: after any MH step on a θ, an MH step on the decision tree is also taken. This holds for both tree inference approaches.

Chapter 5

Experimental Evaluation

In essence, we need to evaluate three aspects. The first aspect is the algorithm that performs inference to find a good policy given a region configuration. We inspect how the algorithm accelerates if we leverage the soft prior knowledge represented as decision trees. The baseline is the same algorithm, but with the trivial region configuration, i.e. no soft prior knowledge is added. Furthermore, we inspect this acceleration in terms of the number of calls to the policy evaluation procedure and in terms of wall-clock time. The second aspect is the generalizability potential of the inferred action distributions per region. The final aspect is the inference of the soft priors (i.e. the region configurations) themselves. These three aspects are evaluated and discussed respectively in Sections 5.2, 5.3, and 5.4. Each of these sections starts by listing the concrete questions that are investigated and answered. Of course, the answers to these questions are limited in scope to the MDPs on which they are evaluated. Before any algorithms are evaluated, the MDP instances and their respective region configurations are defined in Section 5.1. All experiments are conducted on a DelftBlue[10] compute node with 40 CPUs and a maximum of 2GB per CPU. An exception is made when the results of experiments are expressed in terms of wall-clock time: then the number of CPUs is decreased to 1 for a fair comparison.

5.1 MDP Instances

To experimentally evaluate the inference algorithm, the example MDPs described in Section 4.2.1 and one additional MDP are used. For each MDP, three region configurations are investigated. The first, RC 1, is always the trivial region configuration. I.e. $\mathcal{S}$ is considered as one region. RC 1 functions as a baseline. The other two region configurations depend on the MDP instance. As the region configuration number increases, the level of sophistication of the soft prior knowledge increases. This allows us to observe how big the impact of more (sophisticated) soft prior knowledge is on the performance of the inference algorithm.

For ease of reference, each MDP instance is labeled with a short name. The MDP visualized in Figure 4.1 (left) is named `NAV`, as in "navigation". The three region configurations for this MDP that are investigated are RC 1, and the two region configurations given in Figure 4.1 (right).

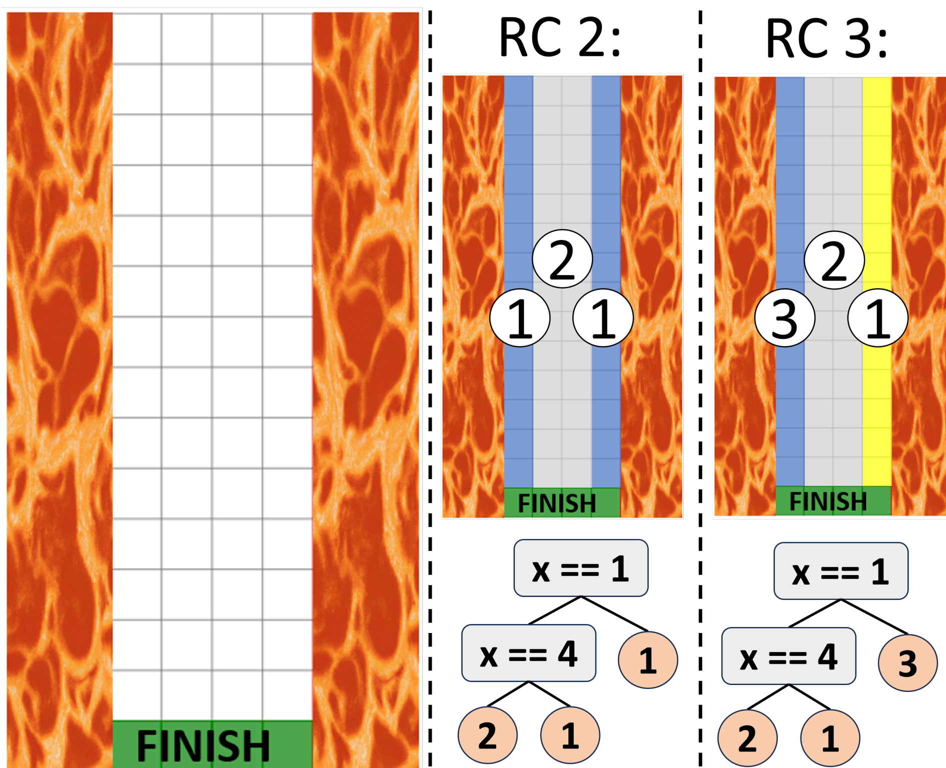

Figure 5.1: the LAVA MDP and its region configurations.

Next, let us inspect the second MDP introduced in Section 4.2.1 (Figure 4.2) and its corresponding region configurations (Figure 4.3). We consider two variants of this MDP. In the first variant, the treasure is locked. In the second variant, the treasure is unlocked. That means that the agent does not need to pick up the key; i.e. the state factor *has_key* is irrelevant considering the goal. We refer to these two variants as LOCK and UNLOCK respectively. Considering these two variants allows us to inspect what happens when we add unjustified soft prior knowledge.

Finally, we introduce an MDP named LAVA. This MDP consists of a pathway with lava lakes on both sides (Figure 5.1 (left)). The available actions are $\{\uparrow, \downarrow, \leftarrow, \rightarrow, IDLE\}$. When the agent reaches the finish line, the agent receives a positive reward. However, when the agent falls into the lava, it receives a large negative reward. Furthermore, there is again a slip probability of 20%. This means that walking downwards or upwards when the agent is directly next to the lava bears a 10% risk of falling into it. Hence, stepping away from the lava - even though this does not bring the agent any closer to the finish - is often beneficial. Again, two region configurations are considered (Figure 5.1 (middle and right)) in addition to the trivial RC 1. RC 2 considers both edges next to the lava as one region, whereas RC 3 considers each edge as a separate region.

The exact sizes of the state and action spaces of each MDP are given in Table 5.1. Note

Table 5.1: Sizes of state and action spaces per example MDP.

MDP	$\|\mathcal{S}\|$	$\|\mathcal{A}\|$
NAV	155	5
LOCK	50	7
UNLOCK	50	7
LOCK_BIG	98	7
UNLOCK_BIG	98	7
LOCK_2	50	7
UNLOCK_2	50	7
LAVA	60	5

that this table also contains variants of the `LOCK` and `UNLOCK` MDPs that will be explained further in Section 5.3. For every MDP: $\gamma = 0.99$.

5.2 Policy Inference Given a Region Configuration

In this section, the algorithm that infers policies given a region configuration is evaluated. There are two use cases of the algorithm: policy search when we do and when we do not have access to the transition model of the environment. These use cases can be referred to as the planning use case and the reinforcement learning use case respectively. For the planning use case, simplified value iteration can be used for policy evaluation. In this case, the acceleration of the algorithm in terms of wall-clock time is informative. In the case when we do not have access to the transition model, policies can be evaluated through sampled trajectories; i.e. let n agents follow the policy for some amount of time steps and see how well they perform on average. This latter policy evaluation strategy can be costly. Hence, for the reinforcement learning use case, the acceleration of the algorithm in terms of the number of policies that are evaluated is informative. In this section, the following questions will be investigated and answered:

Q1 Does the policy inference algorithm accelerate by leveraging soft prior knowledge in the reinforcement learning use case?

Q2 Does the policy inference algorithm accelerate by leveraging soft prior knowledge in the planning use case?

Q3 Can the policy inference algorithm still infer good policies if the soft prior knowledge is unjustified?

Q4 Is putting more effort into inferring the action distributions (i.e. a higher inference balance parameter ψ) always beneficial?

Q5 Does the policy inference algorithm also infer the right action distributions?

Q6 Does the policy inference algorithm also infer the right action distributions, even when the soft prior knowledge is unjustified?

5.2.1 Experimental Setup

The MDPs and region configurations from Section 5.1 are used to evaluate the policy inference algorithm (Section 4.4). The algorithm is initialized with a random policy and is run until the number of calls to the policy evaluation procedure exceeds some threshold. For NAV, this threshold is 1700. For LOCK and UNLOCK, this threshold is 1500, and is 600 for LAVA. The quality of a policy, expressed as $\upsilon \sum_{s \in \mathcal{S}} V^{\pi}(s)$ is tracked over the number of calls to the policy evaluation procedure (**Q1**) and over the wall-clock time (**Q2**). This evaluation is repeated 40 times and the mean policy performances are reported. Furthermore, the standard errors of these means are reported as well. This analysis for the UNLOCK MDP can provide insights to answer **Q3**.

For the rationality parameter υ a value of 1.0 is used. For the hyper-parameter ψ a value of 1000 is used. To answer **Q4**, more values of ψ are investigated for LOCK. For a complete comparison of ψ-values $[0, 100, 1000, 10000]$ alongside υ-values $[1, 5]$, see Appendix B. Recall that the inference balance parameter ψ is the expected number of MH steps on the θs per $|\mathcal{S}|$ steps on π (Section 4.4.3) and that υ is the rationality multiplier (Section 4.3.1).

The most important goal of the inference algorithm is to find a good policy. However, the inference procedure also infers an action distribution θ_i for each region i. To verify that the inferred action distributions correspond to our expectations and to answer **Q5**, the final θs are also averaged and visualized. Furthermore, analyzing this for the UNLOCK MDP can provide insight to answer **Q6**.

5.2.2 Results and Discussion

The progress of the mean policy performance over the number of policy evaluations is visualized for MDPs NAV, LOCK, UNLOCK, and LAVA (Figure 5.2). The ribbon around the means displays the standard error of the mean. Occasionally this ribbon cannot be seen; in those cases, the standard error is as big as or smaller than the thickness of the line representing the mean. Similar plots are shown in Figure 5.3; the difference is that the horizontal axis now contains the wall-clock time rather than the number of policy evaluations.

We can see that generally, a more sophisticated region configuration results in accelerated policy learning in the reinforcement learning use case (Figure 5.2) (**Q1**). From the UNLOCK MDP we can see what happens when unjustified soft prior knowledge is added. Fortunately, we see that the performance does not deteriorate much (**Q3**). Furthermore, we can see that added soft prior knowledge occasionally improves and occasionally deteriorates the performance in the planning use case (Figure 5.3) (**Q2**). Especially the LOCK MDP illustrates nicely how sometimes a more simplistic region configuration can result in faster convergence. This is a clear trade-off that should be considered depending on the use case of the algorithm.

The Influence of ψ (Q4) Next, let us further inspect the influence of the amount of computational effort put into inferring the action distributions. This amount of effort is expressed as ψ; as ψ increases, more effort is put into inferring the θs. These results for the

LOCK MDP can be seen in Figure 5.4 for the number of policy evaluations on the horizontal axis and in Figure 5.5 for the wall-clock time on the horizontal axis.

In the reinforcement learning use case, we can see that more effort into inferring the θs results in faster convergence, the more sophisticated the region configuration. When we now consider the planning use case - we plot the wall-clock time on the horizontal axis - we see that a higher ψ is not always beneficial. Of course, inferring the action distribution takes time. Meanwhile, if we have inferred a good action distribution, we make sampling good policies more likely. There is again a clear trade-off: if we are in the planning use case, it can be beneficial to put *some* effort into inferring the θs, but we should also not overdo it.

Correctness of θs (Q5, Q6) The inferred θs are averaged per region over the 40 repeats of the experiment and are visualized (Figures 5.6-5.14). These action distributions are the results of the same experiments that are shown in Figure 5.2. By looking back at the visualizations of the region configurations (Figures 4.1, 4.3 (right), and 5.1) and inspecting these action distributions we can see that the majority roughly is as expected, with a little bit of noise due to sampling (**Q5**).

For each MDP, we can see that the baseline (RC 1) is already able to infer that *IDLE* is never a good action (Figures 5.6, 5.9, 5.12, and 5.15). Furthermore, we can see that there is a preference for moving to the top-left if the agent does not yet have the key and that there is a preference for moving to the bottom right once the agent has picked up the key (Figure 5.10). Analogously, we can see that the inferred action distributions are roughly equal and both encode a preference for moving towards the bottom right when the state factor *has_key* is irrelevant (Figure 5.13). This illustrates how the algorithm leverages soft prior knowledge but is still able to recover if the soft prior knowledge turns out to be unjustified (**Q6**).

Although the majority of the action distributions capture what one would expect, there are still some unexpected aspects. For example, in Figures 5.11 and 5.14 in region 1 one would expect *PICKUP* to be the clearly dominant action and in region 2 one would expect *OPEN* to be the best. These are in fact the actions with the highest probability respectively. However, we see relatively much noise, as the difference with other actions is not large. We speculate that this is because these regions only comprise a single state.

We can see another unexpected aspect in the UNLOCK MDP when the agent does not have the key, but the treasure is also not locked (Figure 5.14 (region 4)). Here, the actions $\uparrow$ and $\leftarrow$ are more likely than we would expect. A distribution that majorly favors $\downarrow$ and $\rightarrow$, and favors *PICKUP* slightly would be more expected. However, remember that policies that do pick up the key before heading to the treasure also receive the large reward. We suspect that these policies are 'not unlikely enough' due to the high value of γ used: 0.99. To verify this suspicion, an extra experiment is done, where now $\gamma = 0.9$. The inferred action distribution for UNLOCK for region 4 in region configuration 3 is visualized again (Figure 5.18). What we can see is that now that γ is decreased, the probabilities of $\uparrow$ and $\leftarrow$ have indeed also decreased.

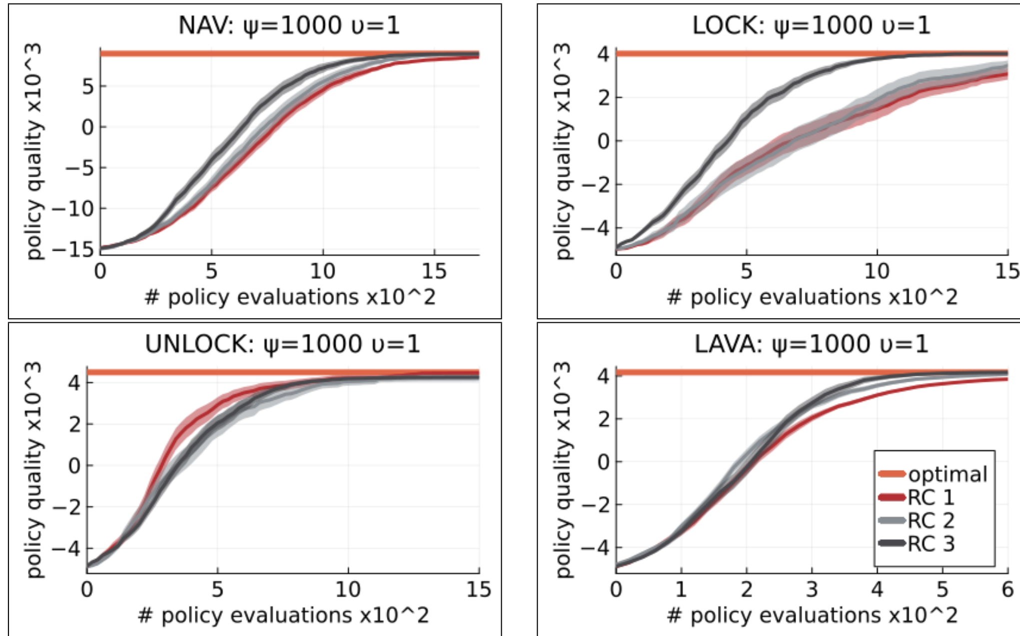

Figure 5.2: Progress of the mean policy quality $(\upsilon\sum_{s\in\mathcal{S}} V^{\pi}(s))$ over the number of policy evaluations for the NAV, LOCK, UNLOCK, and LAVA MDPs for the three different region configurations. The ribbon around the means displays the standard error of the mean.

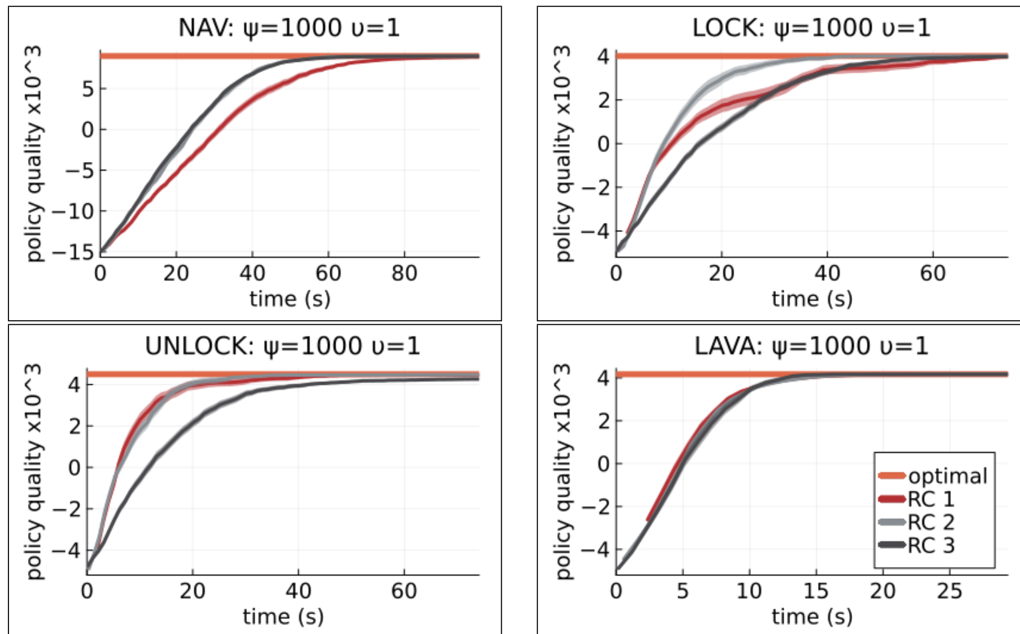

Figure 5.3: Progress of the mean policy quality $(\upsilon\sum_{s\in\mathcal{S}} V^{\pi}(s))$ over the wall-clock time for the NAV, LOCK, UNLOCK, and LAVA MDPs for the three different region configurations. Note that the black and grey lines overlap for NAV.

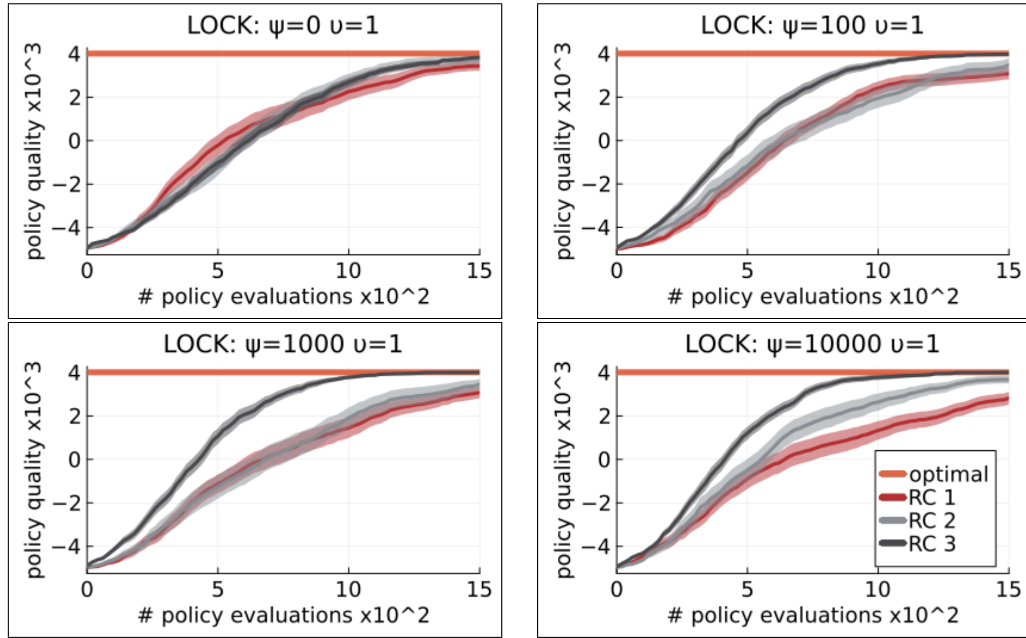

Figure 5.4: Progress of the mean policy quality $(\upsilon \sum_{s \in \mathcal{S}} V^{\pi}(s))$ over the number of policy evaluations for the `LOCK` MDP for the three different region configurations and increasing values of ψ.

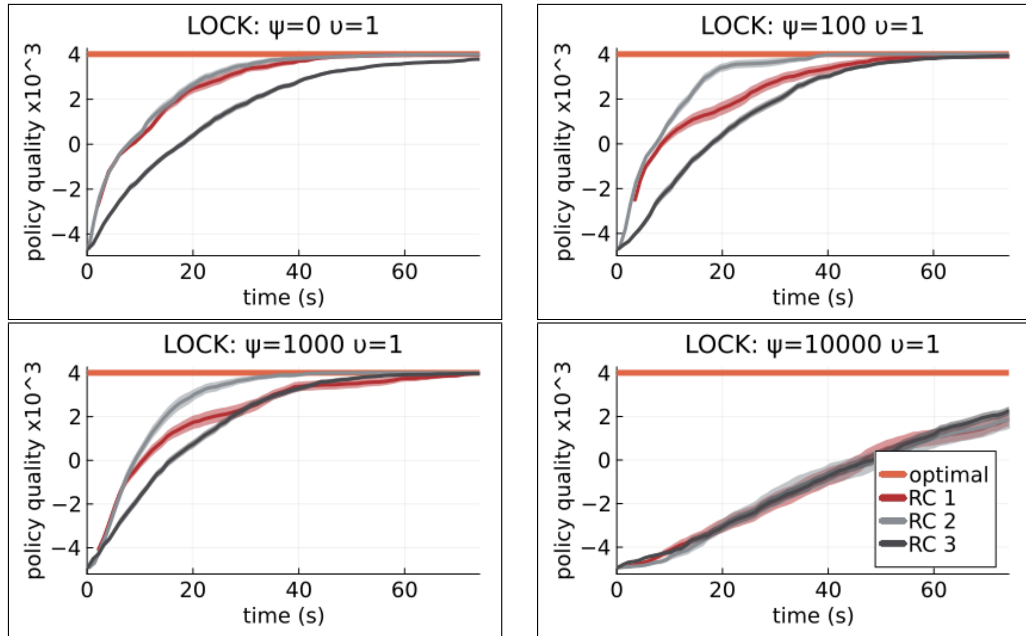

Figure 5.5: Progress of the mean policy quality $(\upsilon \sum_{s \in \mathcal{S}} V^{\pi}(s))$ over the wall-clock time for the `LOCK` MDP for the three different region configurations and increasing values of ψ.

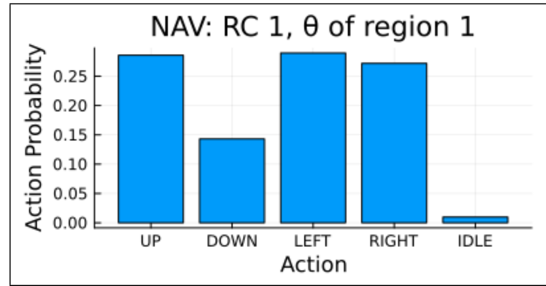

Figure 5.6: Visualization of θ for NAV inferred from region configuration 1. Region 1 is simply $\mathcal{S}$.

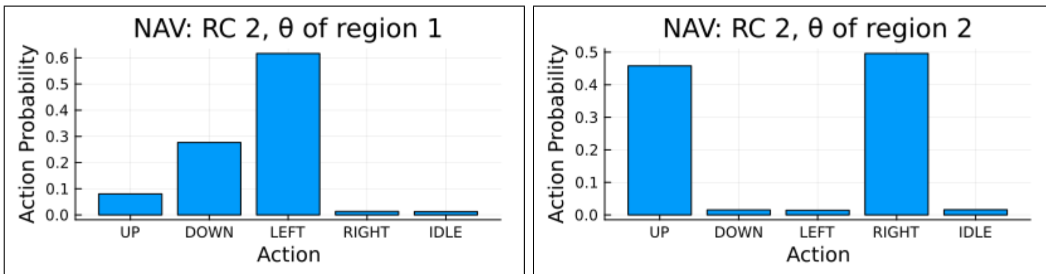

Figure 5.7: Visualization of the θs for NAV inferred from region configuration 2. Region 1 is the "room" in the bottom right and region 2 is the L-shaped "hallway". See Figure 4.1 (right) for a visualization of the regions.

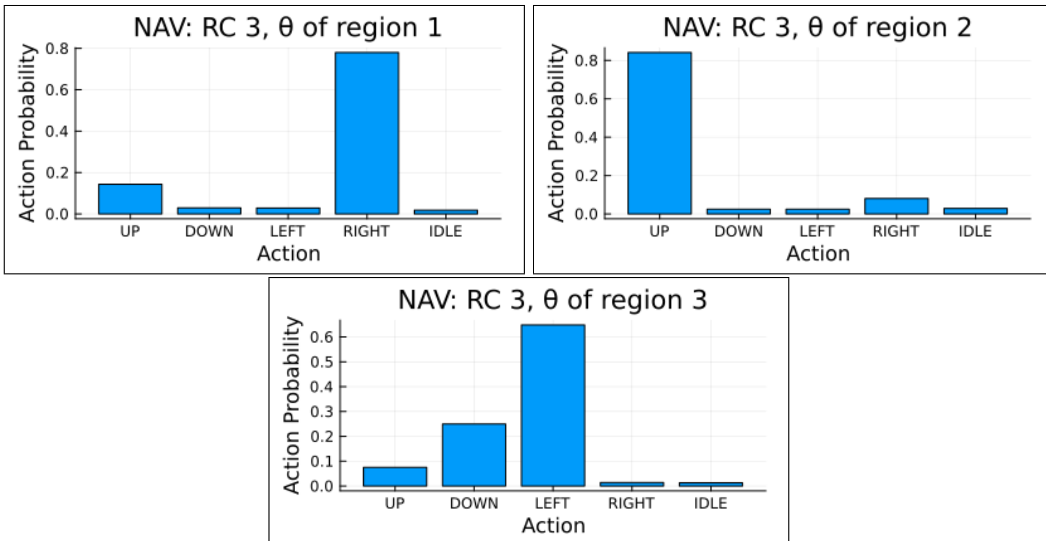

Figure 5.8: Visualization of the θs for NAV inferred from region configuration 3. Regions 1, 2, and 3 are numbered and marked yellow, red, and blue respectively in Figure 4.1 (right).

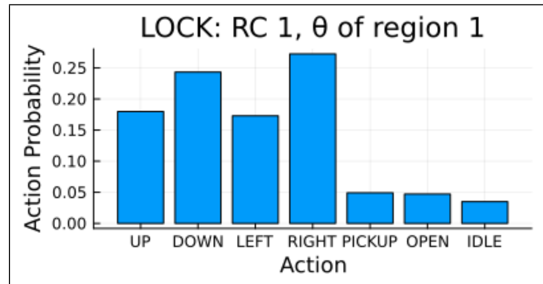

Figure 5.9: Visualization of θ for `LOCK` inferred from region configuration 1. Region 1 is simply $\mathcal{S}$.

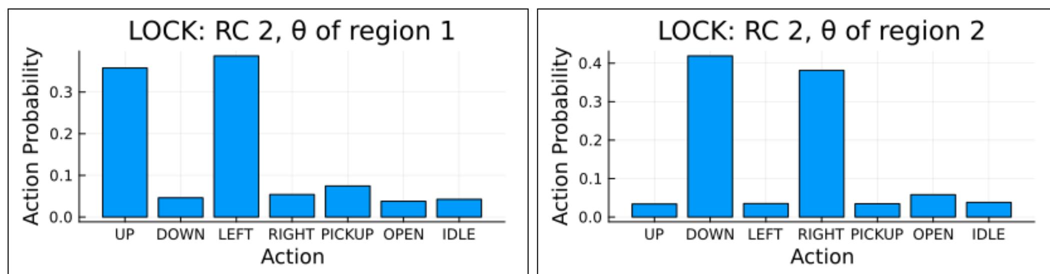

Figure 5.10: Visualization of the θs for `LOCK` inferred from region configuration 2. In region 1 the agent does not yet have the key. In region 2, the agent has picked up the key. See Figure 4.3 for a visualization of the regions.

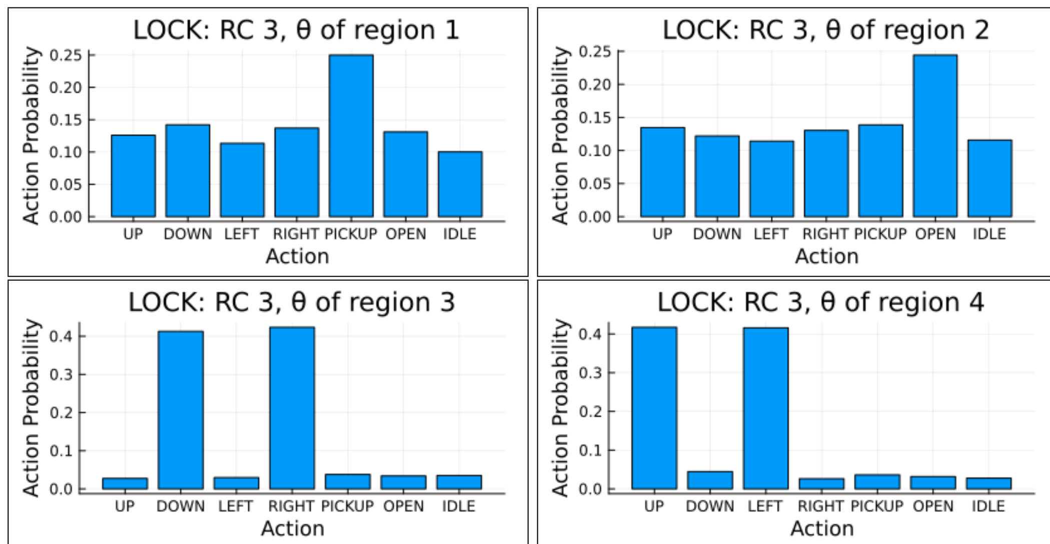

Figure 5.11: Visualization of the θs for `UNLOCK` inferred from region configuration 3. In regions 1 and 2, the agent is at the key or the treasure respectively. In region 4 the agent does not yet have the key. In region 3, the agent has picked up the key. See Figure 4.3 for a visualization of the regions.

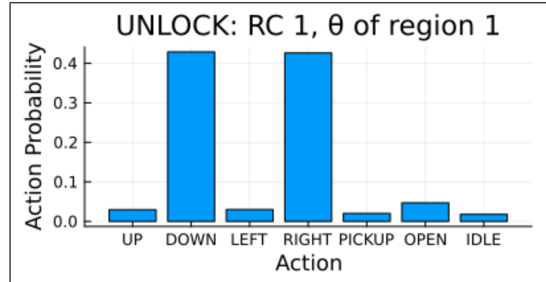

Figure 5.12: Visualization of θ for `UNLOCK` inferred from region configuration 1. Region 1 is simply $\mathcal{S}$.

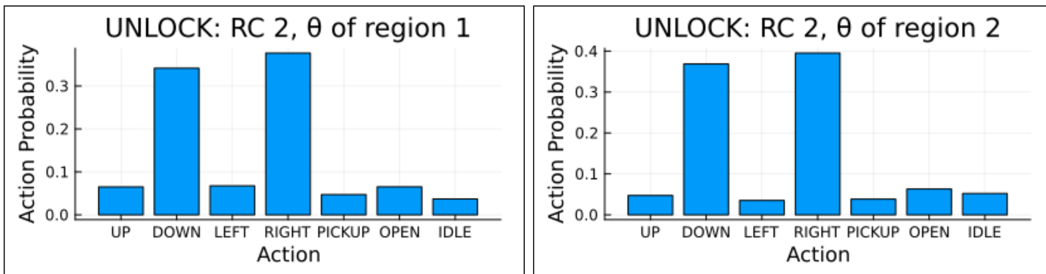

Figure 5.13: Visualization of the θs for `UNLOCK` inferred from region configuration 2. In region 1 the agent does not yet have the key. In region 2, the agent has picked up the key. See Figure 4.3 for a visualization of the regions.

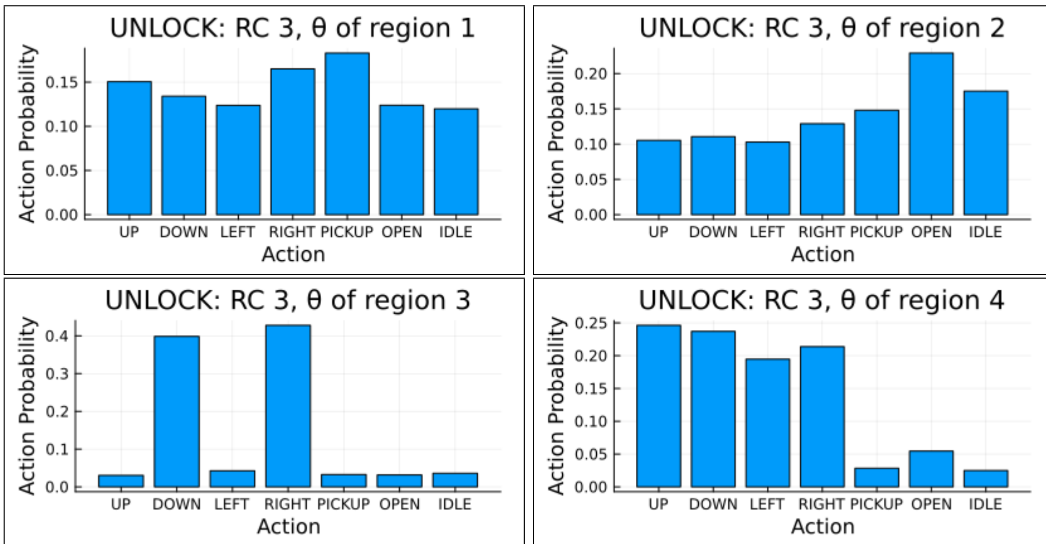

Figure 5.14: Visualization of the θs for `UNLOCK` inferred from region configuration 3. In regions 1 and 2, the agent is at the key or the treasure respectively. In region 4 the agent does not yet have the key. In region 3, the agent has picked up the key. See Figure 4.3 for a visualization of the regions.

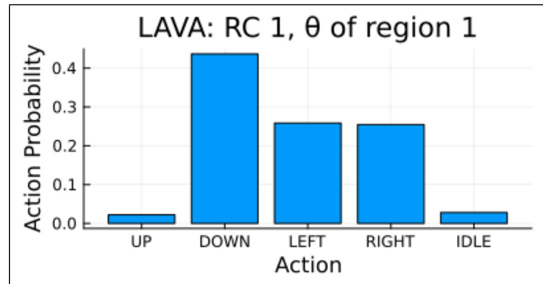

Figure 5.15: Visualization of θ for `LAVA` inferred from region configuration 1. Region 1 is simply $\mathcal{S}$.

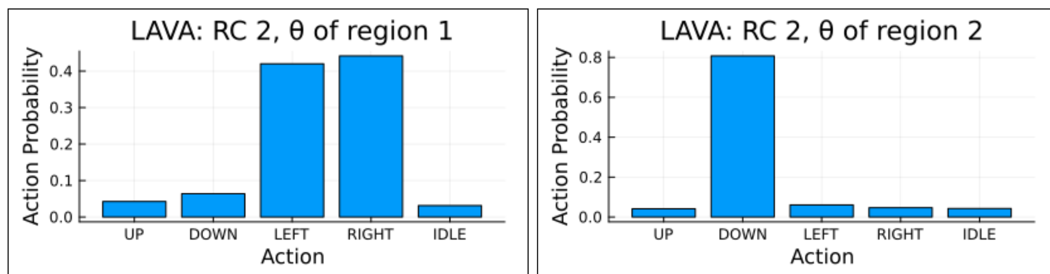

Figure 5.16: Visualization of the θs for `LAVA` inferred from region configuration 2. Region 1 is all the states directly next to the lava and region 2 is the path in the middle. See Figure 5.1 for a visualization of the regions.

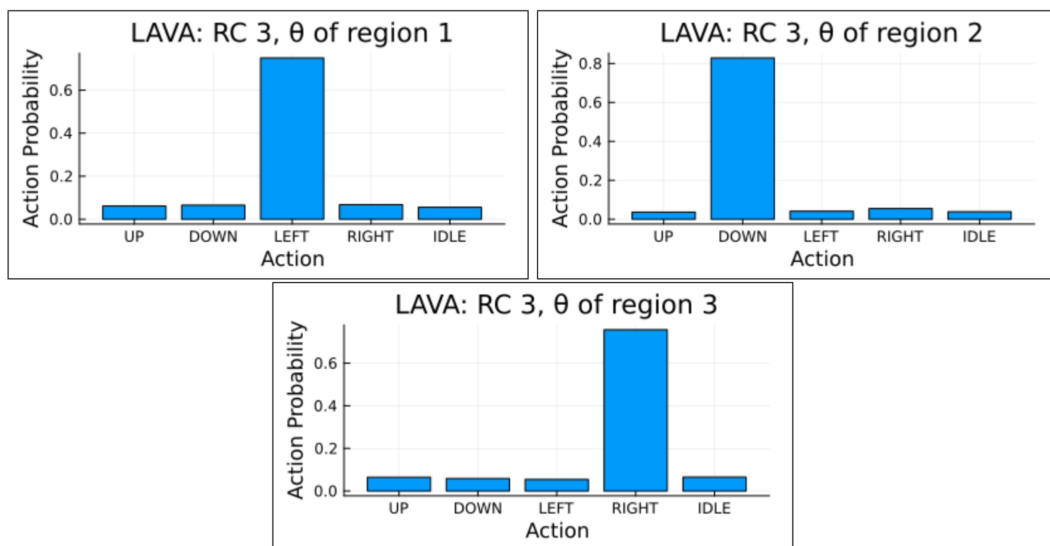

Figure 5.17: Visualization of the θs for `LAVA` inferred from region configuration 3. Region 1 is all the states directly next to the lava on the right side, region 2 is the path in the middle, and region 3 is all the states directly next to the lava on the left side. See Figure 5.1 for a visualization of the regions.

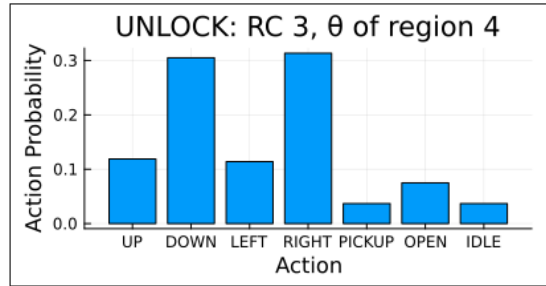

Figure 5.18: Visualization of θ for region 4 of UNLOCK inferred from region configuration 3. $\gamma = 0.9$

5.3 Generalization of θ

In this section, the generalizability potential of the inferred action distributions over regions is assessed. This is done specifically for the LOCK and UNLOCK MDPs. The following questions will be investigated and answered:

Q7 Does initializing the θs with inferred action distributions over regions from the LOCK MDP accelerate the policy search in unseen but similar environments?

Q8 Does initializing the θs with inferred action distributions over regions from the UNLOCK MDP accelerate the policy search in unseen but similar environments?

5.3.1 Experimental Setup

To be able to answer **Q7** and **Q8**, four additional MDPs are introduced: LOCK_BIG, LOCK_2, UNLOCK_BIG, and UNLOCK_2. LOCK_BIG is identical to the LOCK MDP, except the grid is now 7x7 instead of 5x5. UNLOCK_BIG is identical to LOCK_BIG except the key is not needed to open the treasure. Furthermore, LOCK_2 is identical to the LOCK MDP, except the key is replaced from the top-left corner to the top-right corner, and the treasure is replaced from the bottom-right corner to the bottom-left. Finally, the UNLOCK_2 MDP is again identical to LOCK_2 except the key is not needed to open the treasure.

The introduced MDPs are clearly adaptations of the LOCK and UNLOCK MDPs. The inference algorithm from before is now repeated for these adapted MDPs. This is done in two setups:

- θ **init uniform**: the original setup;
- θ **init learned**: the inferred θs from the LOCK and UNLOCK MDPs (as visualized in Figures 5.9 until 5.14) are reused as initialization for the θs of the corresponding regions.

Reusing the inferred action distributions per region can provide an indication of how well the inferred soft prior knowledge generalizes to unseen but similar environments. For this experiment: $\upsilon = 1, \psi = 1000$.

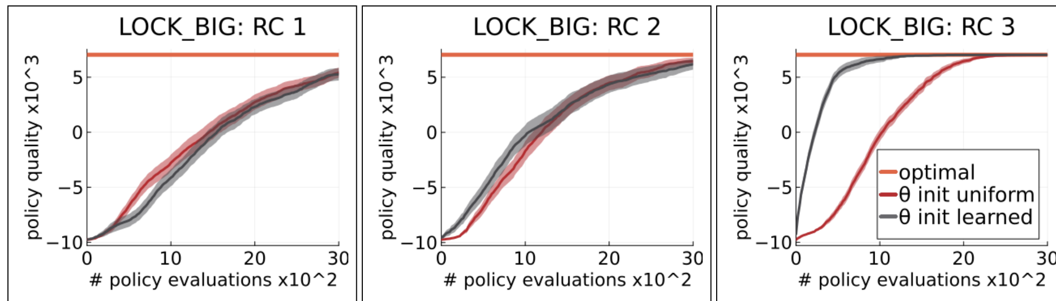

Figure 5.19: Analysis of the generalization of the inferred action distribution from MDP `LOCK` to `LOCK_BIG`.

5.3.2 Results and Discussion

From the results (Figures 5.19, 5.20, 5.21, and 5.22) it can be seen that if the state space is not divided into regions (RC 1), the initialization does not seem to make a difference for how quickly a good policy is found. Likewise, when the state space is divided based on only *has_key*, no significant acceleration can be observed. Finally, we can see that for each MDP with region configuration 3, the learned initialization of the θs accelerates the policy learning. Hence, the situation where the most soft prior knowledge is added has clear merits for generalization to unseen but similar environments (**Q7**, **Q8**).

Interestingly, we can see in Figure 5.21 (RC 3) that the learned initialization very quickly results in good policies compared to the baseline. However, after this steep climb, the average quality of the best policy found seems to stagnate. In the limit, this will still converge to the optimal policy eventually, albeit much later than the baseline. Considering that the θs, in this case, were initialized with the distributions visualized in Figure 5.14, we can phrase a suspicion as to why the algorithm is struggling to find the optimal policy. In region 4 (i.e. the agent does not have the key), an optimal agent would move to the bottom right as the key is not needed to open the treasure. However, the initial action distribution makes policies that *do* move to the key likely as well. The discount factor γ is still 0.99, but as the grid is larger the agent needs to travel more and the drawback of unnecessarily moving to the key is amplified. To overcome this, the θ of region 4 would need to be updated such that probability mass is moved from $\uparrow$ and $\leftarrow$ to $\downarrow$ and $\rightarrow$. As it turns out, overcoming this takes longer than simply initializing the θs uniform.

5.4 Inference of Region Configurations

To evaluate the two tree inference approaches “circumventing stochastic support” and “ancestral sampling” (Section 4.5.2), some experiments are done. The following two questions will be explored and answered:

Q9 Can the “circumventing stochastic support” tree inference approach be used to successfully infer partial trees?

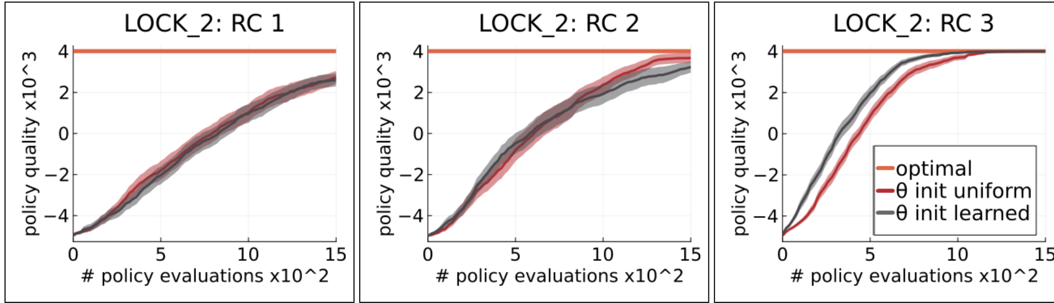

Figure 5.20: Analysis of the generalization of the inferred action distribution from MDP `LOCK` to `LOCK_2`.

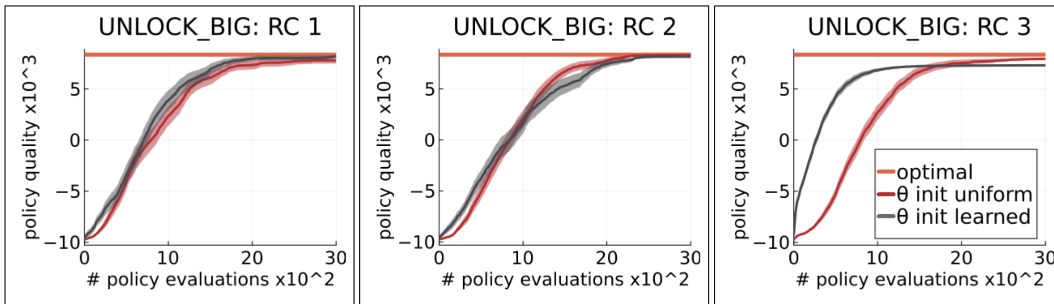

Figure 5.21: Analysis of the generalization of the inferred action distribution from MDP `UNLOCK` to `UNLOCK_BIG`.

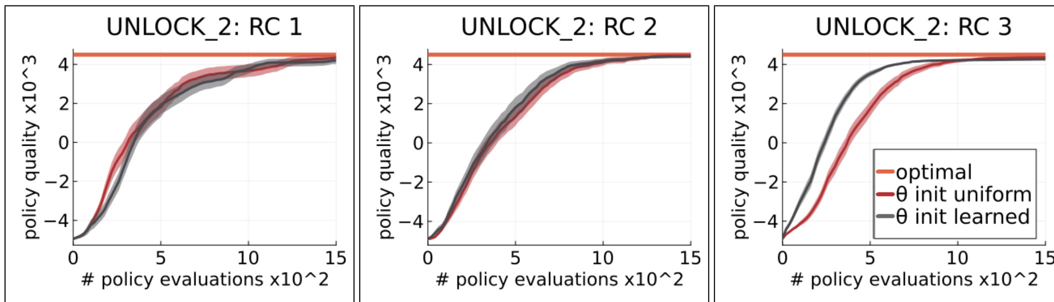

Figure 5.22: Analysis of the generalization of the inferred action distribution from MDP `UNLOCK` to `UNLOCK_2`.

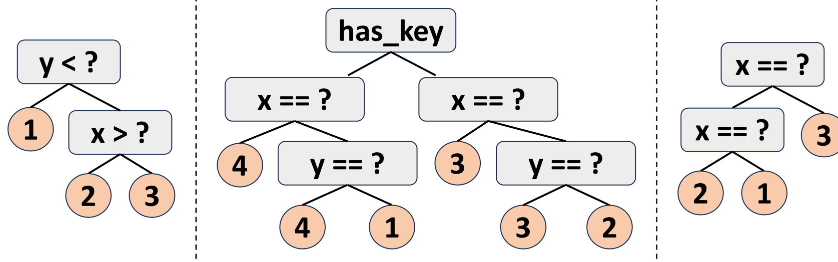

Figure 5.23: Conditioned trees for pivot inference of MDPs `NAV`, `LOCK`, and `LAVA` from left to right respectively.

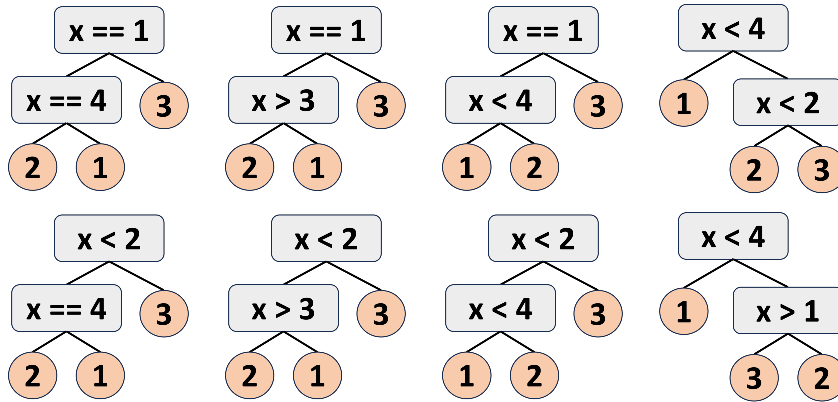

Figure 5.24: Eight equivalent decision trees over the `LAVA` MDP.

Q10 Can the "ancestral sampling" tree inference approach be used to successfully infer trees from scratch?

5.4.1 Experimental Setup

The two tree inference approaches "circumventing stochastic support" and "ancestral sampling" (Section 4.5.2) are evaluated by running the inference algorithms to infer (parts of) the region configurations of MDPs `NAV`, `LOCK`, and `LAVA`. To evaluate the "circumventing stochastic support" approach and to answer **Q9**, the entire decision trees are conditioned except for the pivot values (Figure 5.23). The pivot values considered 'correct' are those of the corresponding trees used in previous experiments, given in Figures 4.1 (RC 3), 4.3 (RC 3), and 5.1 (RC 3) for MDPs `NAV`, `LOCK`, and `LAVA` respectively. Occasionally, some leniency is allowed for these pivots. Specifically, for the `NAV` MDP 12 and 11 are acceptable y-pivots, and 5 and 6 are acceptable x-pivots. Likewise, for the `LAVA` MDP the pivots have to be 1 and 4 but their location can be swapped. The inference algorithm is repeated 100 times for various maximum number of policy evaluations and for $\upsilon = \{1,5\}$.

To evaluate the "ancestral sampling" approach and to answer **Q10**, no parts of the tree

are conditioned and the entire tree needs to be inferred. One aspect that needs to be taken into account is that different trees can represent the same region configuration. Firstly, the most obvious aspect of this is the permutation invariance of the region labels; any permutation of the labels represents the same region configuration as it is just a naming convention. Secondly, different pivots and/or different tree structures can also represent the same region configuration. To illustrate this, eight equivalent trees over the `LAVA` MDP are visualized in Figure 5.24. Many more equivalent trees can be constructed[1]. Note that these trees are *empirically equivalent* for the state space of the `LAVA` MDP; the story would have been different had there been states for which x is smaller than 1 or larger than 4. This empirical equivalence of decision trees is taken into account for the tree inference by iterating over all pairs of states, and verifying that two trees agree about whether the two states belong to the same region. Any pair of states for which the trees disagree is a counterexample, meaning that the trees are not empirically equivalent. If no counterexample can be found, the trees are considered empirically equivalent.

Another aspect to consider for the tree inference is the parameters of the generative model *max_num_regions* and *max_tree_depth*. These values need to be defined beforehand. For the `NAV` and the `LAVA` MDPs, the values are set to *max_num_regions* $= 3$ and *max_tree_depth* $= 2$. The trees considered correct are again those from Figures 4.1 (RC 3) and 5.1 (RC 3) respectively, or any empirically equivalent tree. Again, the same leniency in pivot values as before is allowed. In contrast, for the `LOCK` MDP two setups are evaluated: one where *max_num_regions* $= 2$ and *max_tree_depth* $= 1$ and one where *max_num_regions* $= 4$ and *max_tree_depth* $= 3$. For these two setups, RC 2 and RC 3 of `LOCK` (Figure 4.3) are considered the correct trees respectively. The inference algorithm is repeated 10 times for various maximum number of policy evaluations and for $\upsilon = \{1, 5\}$. For all experiments in this section $\psi = 1000$.

5.4.2 Results and Discussion

The pivot inference algorithm is repeated 100 times and the number of times the inferred tree is correct is reported (Table 5.2). From this, we can see that the algorithm manages to infer the correct pivots for all experiments, given that the maximum number of policy evaluations is large enough (**Q9**). There is also no substantial difference visible between $\upsilon = 1$ and $\upsilon = 5$. The observed difference is likely a result of the random sampling.

Furthermore, the inference algorithm for complete trees is repeated 10 times and the number of times the inferred tree is correct is reported (Table 5.3). These results indicate that the algorithm works well on the `LAVA` MDP and that the success rate of the algorithm again grows with the maximum number of policy evaluations. Inspecting the case of the `LOCK` MDP a bit more, we see that the algorithm is able to infer that a split on `has_key` is beneficial in the $d = 1$ case. However, in the $d = 3$ case, the algorithm again consistently infers a decision tree that only splits on `has_key` and nothing else. This yields a success rate of 0 as it never identifies the location of the key or the location of the treasure as a separate

[1] In total there are 108. There are 3 possible decision nodes to single out the `"x = 1"`-column, 3 possible decision nodes to single out the `"x = 4"`-column times, 2 ways to combine these decision nodes, 6 permutations of the region labels. $3 \times 3 \times 2 \times 6 = 108$

region. This can have two possible reasons. First of all, it is simply unlikely that a correct tree of depth 3 is sampled by forward sampling from the generative model. Secondly, it might be the case that identifying these regions is simply not worth it as the regions consist of only one state. After all, a slight preference for shallow trees is encoded (Section 4.3.2); the single-state regions might not provide enough advantage to overcome this preference. A hint for this second interpretation is seen in Figure 5.11: the action distributions in the single-state regions are spread much more than the action distributions of other regions. The results suggest that the "ancestral sampling" approach works well for simple trees but struggles as the trees get deeper and more sophisticated (**Q10**).

Table 5.2: Results for "circumventing stochastic support" (where only pivots are inferred). Reported are the number of correctly inferred trees out of 100 repeats of the algorithm.

max policy	NAV		LOCK		LAVA	
evaluations	$\upsilon = 1$	$\upsilon = 5$	$\upsilon = 1$	$\upsilon = 5$	$\upsilon = 1$	$\upsilon = 5$
500	7	9	6	2	49	40
1000	39	25	24	19	69	75
2000	78	76	71	62	89	92
4000	98	98	97	99	98	92
6000	100	100	100	100	100	97

Table 5.3: Results for "ancestral sampling" (where complete trees are inferred). Reported are the number of correctly inferred trees out of 10 repeats of the algorithm. d indicates the max_tree_depth parameter.

max policy	NAV		LOCK (d=1)		LOCK (d=3)		LAVA	
evaluations	$\upsilon = 1$	$\upsilon = 5$	$\upsilon = 1$	$\upsilon = 5$	$\upsilon = 1$	$\upsilon = 5$	$\upsilon = 1$	$\upsilon = 5$
500	0	0	7	3	0	0	6	1
1000	0	0	9	9	0	0	7	10
2000	2	0	10	10	0	0	10	10
4000	4	5	10	10	0	0	10	10
6000	4	6	10	10	0	0	10	10

Chapter 6

Conclusions and Future Work

6.1 Conclusions and Limitations

This thesis has introduced the idea of prior action distributions over regions of the state space of MDPs, which are leveraged to accelerate Bayesian policy search. The region configurations are represented as decision trees over state factors. A domain expert can define how the state space should be divided into regions, this division is then leveraged to find good policies sooner than when no division into regions is used. This works because action distributions over regions are inferred *while* a policy is inferred. As a result, when a good action is found in a region, that action will automatically become more likely in the rest of the region. Similarly, when an action is found to often yield low or even negative reward, that action will automatically become less likely in the rest of the region.

The algorithm can work for two use cases: policy search when we do and when we do not have access to the transition model of the environment. These use cases are referred to as the planning use case and the reinforcement learning use case respectively. For the planning use case, the acceleration of the algorithm in terms of the wall-clock time is informative. Meanwhile, for the reinforcement learning use case, the acceleration of the algorithm in terms of the number of policies that are evaluated is informative as empirically evaluating a policy can be expensive. On the one hand, more sophisticated region configurations show acceleration for the reinforcement learning use case for all MDPs that are evaluated. On the other hand, the experiments indicate that informative region configurations that do not divide the state space into too many regions lead to improved performance in the planning use case. This is a clear trade-off that should be considered depending on the use case of the algorithm.

Occasionally a domain expert could make a mistake and provide an unjustified region configuration. For example, dividing the state space by whether or not the agent is in possession of a key when it turns out this key is not needed. It is demonstrated that given such an incorrect region configuration, the algorithm still manages to infer (near) optimal policies eventually. In some cases, the inferred action distributions over regions can even be useful for generalization to unseen but similar MDPs. In other cases, the inferred action distributions do not generalize well at all.

Other than having a domain expert specify the region configuration, we also investigated initial inference strategies for inferring these decision trees themselves. Two setups are evaluated. In the first setup, the structure of the tree is given and only the pivots of the decision nodes are inferred. In the second setup, the entire tree is inferred from scratch. The first setup is shown to work successfully for all evaluated MDP instances, given that we run the algorithm long enough. The second setup also provides promising results as it successfully infers small trees, but is unsuccessful for more sophisticated region configurations.

Finally, let us highlight two more limitations of this work. Firstly, a limitation is that multiple (hyper)parameters are introduced. Guessing appropriate values for the (hyper)parameters might be challenging, but executing an extensive (hyper)parameters optimization procedure might not be worth it for certain problems. This is a trade-off that is further addressed in Section 6.2. Secondly, it is worth considering that the leveraging of a division of $\mathcal{S}$ into regions would not work for all possible MDPs. For example, the state space might be exceptionally small or there might be no natural way to batch states that are similar towards a certain goal. Hence, the methodology proposed in this thesis is not a one-size-fits-all approach.

6.2 Future Work

The insights provided by this thesis open many potential directions for future research. Five interesting directions are outlined in this section.

Further Analysis of Parameters As mentioned before, this work introduces multiple (hyper)parameters. Although the inference balance parameter ψ (Section 4.4.3) and the rationality parameter υ (Section 4.3.1) are investigated in Chapter 5 and Appendix B, the precision of the action simplex random walk (Section 4.4.2) is not optimized nor investigated. Optimizing and investigating the impact of this precision parameter potentially accelerates inference. Furthermore, for υ only the values 1 and 5 are evaluated. Higher values of υ break the algorithm since the `factor` functionality is not natively built into Gen (Section 4.4.3). However, when the maximum reward is lower than 100, higher values of υ can be evaluated. Moreover, the `is_leaf` Bernoulli parameter as described in Section 4.3.2 can also be refactored as a hyperparameter and optimized.

Other than that, the *max_depth* and *max_num_regions* parameters of the decision tree need to be known before inference can start. Since these are *maximum* values, they can be set preposterously high in the case of uncertainty. Future work could evaluate the potential deterioration of the algorithm when these bounds are defined too loosely.

Continuous States and/or Actions The methodology of this thesis assumes a discrete set of states and a discrete set of actions. Real-world problems, however, often contain for example continuous sensor readings or continuous decisions: e.g. how fast should we go or how much force should we apply. The real world is simply not nicely divided in a grid. One simple strategy to deal with this is the discretization of the state and action space. Another strategy is to extend this thesis to work directly on continuous MDPs.

Furthermore, the state and action spaces of the MDPs evaluated in this thesis have a maximum size of 155 and 7 respectively. Future work can explore the idea of prior action distributions over regions with larger state and action spaces. The work of Fernandes and Atchley[12] might help for the efficiency of the random walk over the action simplex (Section 4.4.2).

Extension to POMDPs In Markov Decision Processes, it is assumed that the agent can observe exactly what the state of the (agent in the) world is. This assumption is false in many real-world scenarios, for example when the agent has sensors of limited quality. It is more realistic to assume that there is some observation that provides information about the current state but without 100% certainty. These cases can be modeled with Partially Observable Markov Decision Processes (POMDPs) [18]. An extension of this thesis to POMDPs has the potential to solve more realistic real-world problems.

More Sophisticated Tree Inference The current approach for unconditioned tree inference is relatively simplistic; if the correct tree is never generated through forward sampling from the generative model, it is never inferred correctly. Perhaps the most interesting direction of future work is investigating more sophisticated tree inference algorithms. Possible approaches apply, and accept or reject, updates to a current tree rather than proposing a completely new tree in each step. Growing a decision tree by first inferring the root and repeatedly inferring the rest of the tree is also a strategy worth investigating. Instead of such a top-down approach, a bottom-up approach where states are repeatedly merged to form regions might also be worth exploring.

A valuable observation might be that a good region configuration identifies regions in which actions are similar. In other words, the *entropy* of action distributions over regions is low. Hence, the entropy of the action distributions might be a useful measure to guide a complex tree inference procedure. This can be used as a proxy for which tree is better than other trees, but also as guidance for what part of a tree to update.

Pushing the Limits of Tree Inference The methodology explained and evaluated in this thesis leverages a decision tree that divides the state space into regions, to find good policies sooner than when no division into regions is used. The policies themselves, however, are still defined enumeratively as a dictionary that maps states to actions. For each region, an action distribution is inferred which is meanwhile used to infer the policy. By continuing until the action distributions are each completely concentrated on a single action, the policy itself can be represented as a decision tree. A great merit of this is that policies represented as a decision tree are interpretable for humans[25]. Furthermore, a policy represented as a (size-limited) decision tree will likely occupy less memory than a policy represented as a dictionary. Perhaps a size-limited decision tree as a policy does not always gain as much reward as the optimal policy. However, one can imagine use cases when one is willing to trade off a little bit of performance for a more compact and understandable representation of the policy.

6.3 Broader Implications

A potential broader implication of this thesis is the creation of artificially intelligent agents that exhibit thought processes closer to those of humans. The capability to divide the state space into regions can be interpreted as generating an abstract representation of the problem. Being able to reason abstractly about problems makes AIAs 'think' more like humans. Furthermore, when a human wonders why an AIA made a certain decision, this abstract representation of the state space might provide meaningful insight. Additionally, by pushing the boundaries of this research, there is the prospect of completely eliminating the enumerative policy representation and representing policies as decision trees for some MDPs. This can also make the behavior of AIAs more interpretable and understandable for humans.

Bibliography

[1] Hagai Attias. Planning by probabilistic inference. In Christopher M. Bishop and Brendan J. Frey, editors, *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics*, volume R4 of *Proceedings of Machine Learning Research*, pages 9–16. PMLR, 2003. URL `https://proceedings.mlr.press/r4/attias03a.html`. Reissued by PMLR on 01 April 2021.

[2] Richard Ernest Bellman. *Markovian Decision Processes*, chapter 11, pages 317–330. Princeton University Press, 1957. URL `https://gwern.net/doc/statistics/decision/1957-bellman-dynamicprogramming.pdf`.

[3] Christopher M. Bishop. Probability distributions. In *Pattern Recognition and Machine Learning*, chapter 2, pages 76–78. Springer, first edition, 2006. ISBN 978-0-387-31073-2.

[4] Matthew Botvinick and Marc Toussaint. Planning as inference. *Trends in Cognitive Sciences*, 16(10):485–488, 2012. ISSN 1364-6613. doi: https://doi.org/10.1016/j.tics.2012.08.006. URL `https://www.sciencedirect.com/science/article/pii/S1364661312001957`.

[5] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1):49–107, 2000. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(00)00033-3. URL `https://www.sciencedirect.com/science/article/pii/S0004370200000333`.

[6] Leo Breiman. *Classification and Regression Trees*. Routledge, first edition, 1984. ISBN 9781315139470.

[7] Thiago P. Bueno, Denis D. Mauá, Leliane N. de Barros, and Fabio G. Cozman. Markov decision processes specified by probabilistic logic programming: Representation and solution. In *2016 5th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 337–342, 2016. doi: 10.1109/BRACIS.2016.068.

[8] Elena Congeduti and Frans A. Oliehoek. A cross-field review of state abstraction for markov decision processes. In *34th Benelux Conference on Artificial Intelligence*

(BNAIC) and the 30th Belgian Dutch Conference on Machine Learning (Benelearn), 2022. URL https://bnaic2022.uantwerpen.be/wp-content/uploads/BNAICBeNeLearn_2022_submission_3386.pdf.

[9] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: A general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 221–236, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314642. URL http://doi.acm.org/10.1145/3314221.3314642.

[10] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 1). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.

[11] Owain Evans, Andreas Stuhlmüller, John Salvatier, and Daniel Filan. Modeling agents with probabilistic programs. http://agentmodels.org, 2017. Accessed: 2023-11-21.

[12] Andrew D. Fernandes and William R. Atchley. Site-specific evolutionary rates in proteins are better modeled as non-independent and strictly relative. *Bioinformatics (Oxford, England)*, 24(19):2177–2183, 2008. ISSN 1367-4803. doi: 10.1093/bioinformatics/btn395. URL https://doi.org/10.1093/bioinformatics/btn395.

[13] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artificial Intelligence*, 57(2):323–389, 1992. ISSN 0004-3702. doi: 10.1016/0004-3702(92)90021-O. URL https://www.sciencedirect.com/science/article/pii/000437029290021O.

[14] Noah D. Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2023-6-19.

[15] Noah D. Goodman, Joshua B. Tenenbaum, and The ProbMods Contributors. Probabilistic Models of Cognition. http://probmods.org/, 2016. Accessed: 2023-12-6.

[16] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Future of Software Engineering Proceedings*, FOSE 2014, page 167–181, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328654. doi: 10.1145/2593882.2593900. URL https://doi.org/10.1145/2593882.2593900.

[17] Nicholas K. Jong and Peter Stone. State abstraction discovery from irrelevant state variables. In *IJCAI*, volume 8, pages 752–757, 2005.

[18] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(98)00023-X. URL https://www.sciencedirect.com/science/article/pii/S000437029800023X.

[19] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. *Machine Learning*, 49 (2):193–208, 2002. ISSN 1573-0565. doi: 10.1023/A:1017932429737.

[20] Bret Larget and Donald L. Simon. Markov Chain Monte Carlo Algorithms for the Bayesian Analysis of Phylogenetic Trees. *Molecular Biology and Evolution*, 16(6): 750–750, 1999. ISSN 0737-4038. doi: 10.1093/oxfordjournals.molbev.a026160. URL https://doi.org/10.1093/oxfordjournals.molbev.a026160.

[21] Sergey Levine. Reinforcement learning and control as probabilistic inference: Tutorial and review. *CoRR*, abs/1805.00909, 2018. URL http://arxiv.org/abs/1805.00909.

[22] Lihong Li, Thomas J. Walsh, and Michael L. Littman. Towards a unified theory of state abstraction for mdps. *AI&M*, 2006.

[23] Andrew Kachites McCallum. *Reinforcement learning with selective perception and hidden state*. University of Rochester, 1996.

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

[25] Christoph Molnar. Interpretable machine learning. https://christophm.github.io/interpretable-ml-book/, 2023. Accessed: 2023-11-24.

[26] Davide Nitti, Vaishak Belle, and Luc De Raedt. Planning in discrete and continuous markov decision processes by probabilistic programming. In Annalisa Appice, Pedro Pereira Rodrigues, Vítor Santos Costa, João Gama, Alípio Jorge, and Carlos Soares, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 327–342, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23525-7. URL https://doi.org/10.1007/978-3-319-23525-7_20.

[27] Rafael Rodriguez-Sanchez, Benjamin Adin Spiegel, Jennifer Wang, Roma Patel, Stefanie Tellex, and George Konidaris. RLang: A declarative language for describing partial world knowledge to reinforcement learning agents. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 29161–29178. PMLR, 2023. URL https://proceedings.mlr.press/v202/rodriguez-sanchez23a.html.

[28] Sheldon M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists*. Elsevier Academic Press, fourth edition, 2009. ISBN 978-0-12-370483-2.

[29] Stuart Russell and Peter Norvig. Probabilistic reasoning over time. In *Artificial Intelligence A Modern Approach*, chapter 15, pages 566–570. Pearson Education, third edition, 2010. ISBN 978-0-13-604259-4.

[30] Stuart Russell and Peter Norvig. Probabilistic reasoning. In *Artificial Intelligence A Modern Approach*, chapter 14, pages 510–529. Pearson Education, third edition, 2010. ISBN 978-0-13-604259-4.

[31] Stuart Russell and Peter Norvig. Making complex decisions. In *Artificial Intelligence A Modern Approach*, chapter 17, pages 645–658. Pearson Education, third edition, 2010. ISBN 978-0-13-604259-4.

[32] Rolf A. N. Starre, Marco Loog, Elena Congeduti, and Frans A. Oliehoek. An analysis of model-based reinforcement learning from abstracted observations. *Transactions on Machine Learning Research*, 2023.

[33] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction*. The MIT Press, second edition, 2018. ISBN 978-0-262-19398-6.

[34] Ingo Thon, Bernd Gutmann, and Guy Van Den Broeck. Probabilistic programming for planning problems. In *Proceedings of the 6th AAAI Conference on Statistical Relational Artificial Intelligence*, AAAIWS'10-06, page 98–99. AAAI Press, 2010. doi: 10.5555/2908567.2908586.

[35] Marc Toussaint and Amos Storkey. Probabilistic inference for solving discrete and continuous state markov decision processes. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 945–952, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933832. doi: 10.1145/1143844.1143963. URL https://dl.acm.org/doi/10.1145/1143844.1143963.

[36] Marc Toussaint, Amos Storkey, and Stefan Harmeling. *Expectation-Maximization methods for solving (PO) MDPs and optimal control problems*. Cambridge University Press, 2010. ISBN 9780511984679. doi: 10.1017/CBO9780511984679.019.

[37] Jan-Willem van de Meent, Brooks Paige, David Tolpin, and Frank Wood. Black-box policy search with probabilistic programs. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 1195–1204, Cadiz, Spain, 2016. PMLR. URL https://proceedings.mlr.press/v51/vandemeent16.html.

[38] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming, 2021. URL https://arxiv.org/pdf/1809.10756.pdf.

[39] David Wingate, Noah D. Goodman, Daniel M. Roy, Leslie P. Kaelbling, and Joshua B. Tenenbaum. Bayesian policy search with policy priors. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, page 1565–1570. AAAI Press, 2011. ISBN 9781577355144. doi: 10.5555/2283516.2283656.

[40] Ryan Yang, Tom Silver, Aidan Curtis, Tomas Lozano-Perez, and Leslie Kaelbling. Pg3: Policy-guided planning for generalized policy generation. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 4686–4692. International Joint Conferences on Artificial Intelligence Organization, 2022. doi: 10.24963/ijcai.2022/650. URL `https://doi.org/10.24963/ijcai.2022/650`. Main Track.

Appendix A

Glossary

In this appendix we provide an overview of frequently used abbreviations:

MDP: Markov Decision Process

AIA: Artificially Intelligent Agent

VI: Value Iteration

MCMC: Markov Chain Monte Carlo

MH: Metropolis-Hastings

RC: Region Configuration

POMDP: Partially Observable Markov Decision Process

Appendix B

Hyperparameter Analysis

In this appendix, we study the effect of the hyperparameters ψ and υ. Recall that ψ is the expected number of MH updates on θ per $|\mathcal{S}|$ updates on π (Section 4.4.3); i.e. the amount of computational effort that is put into inferring the θs. Furthermore, recall that υ is the rationality multiplier (Section 4.3.1).

The progress of the mean policy performance over the number of policy evaluations is visualized in Figures B.1, B.3, B.5, and B.7 for MDPs `NAV`, `LOCK`, `UNLOCK`, and `LAVA` respectively. Likewise, the progress of the mean policy performance over the wall-clock time is visualized in Figures B.2, B.4, B.6, and B.8. The ribbon around the means display the standard error of the mean. Occasionally this ribbon cannot be seen; in those cases the standard error is as big as or smaller than the thickness of the line representing the mean. Each figure visualizes this for various values of ψ and υ to inspect the influence of these hyperparameters. Note that $\psi = 0$ means that *none* of the MH steps perform an update to a θ. In other words, the prior remains uniform no matter the region configuration. The results are consistent with the discussion provided in Section 5.2.2.

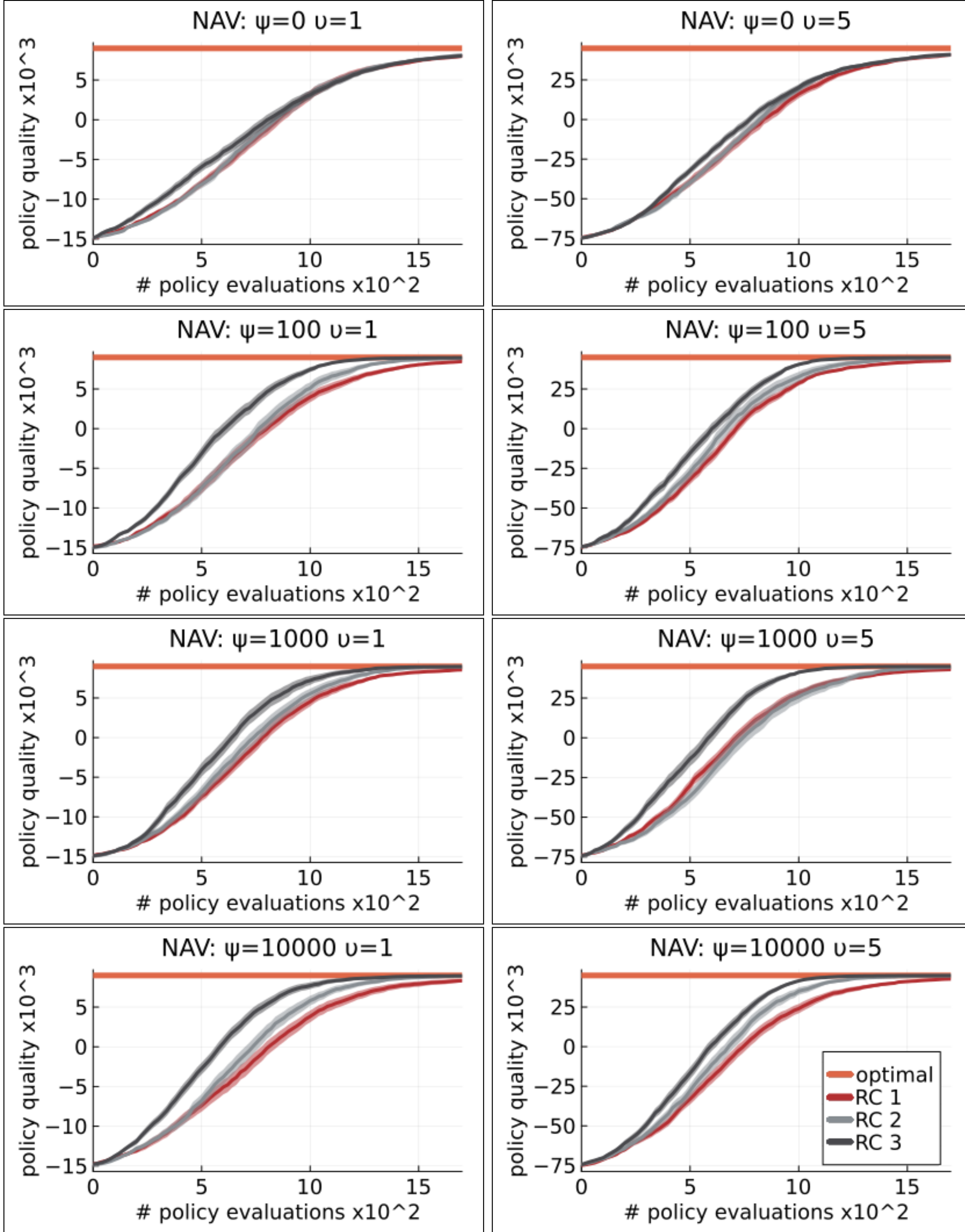

Figure B.1: Progress of the mean policy quality $(\upsilon \sum_{s \in \mathcal{S}} V^{\pi}(s))$ over the number of policy evaluations for the `NAV` MDP for the three different region configurations and various values of ψ and υ.

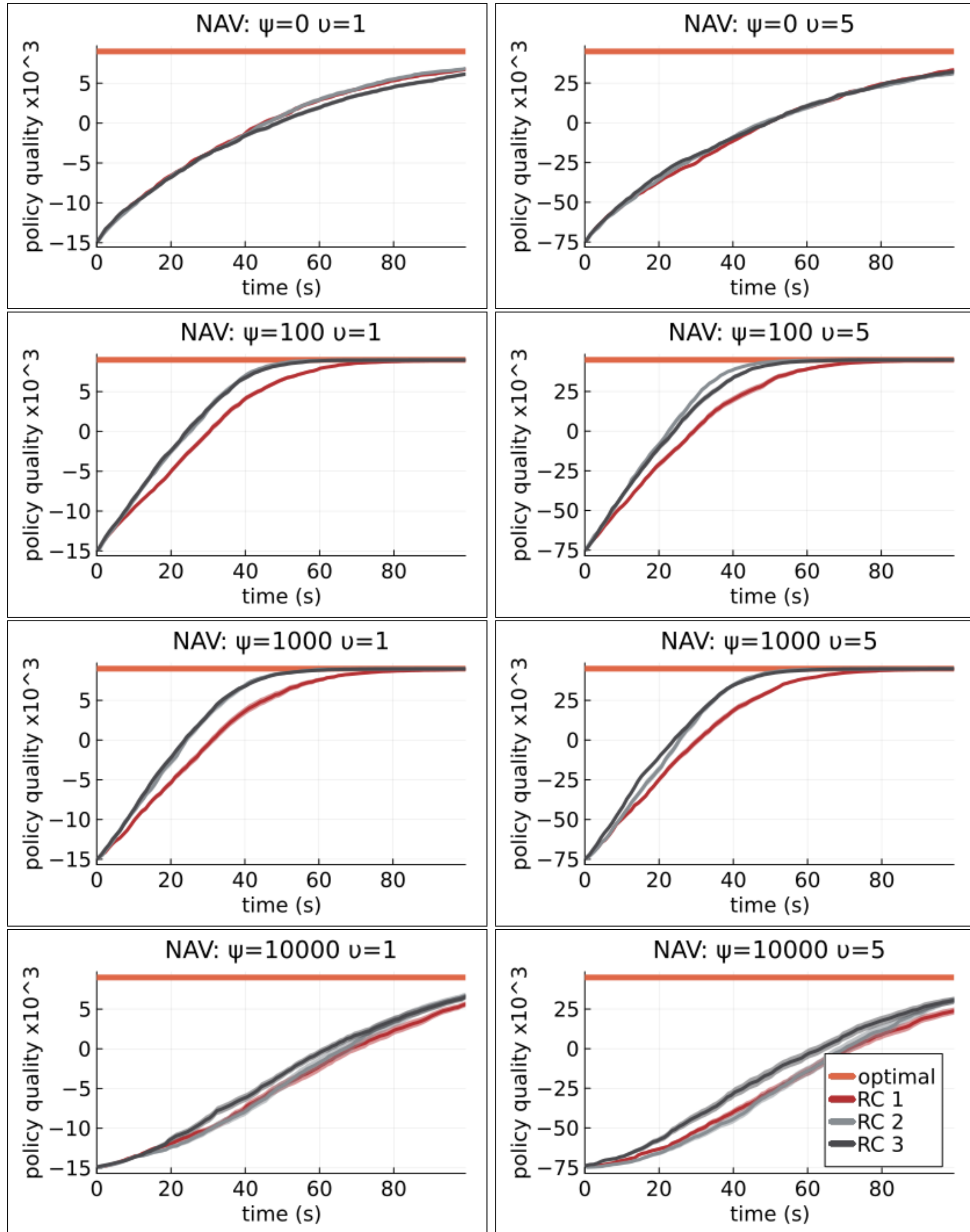

Figure B.2: Progress of the mean policy quality $(\upsilon \sum_{s \in \mathcal{S}} V^{\pi}(s))$ over the wall-clock time for the NAV MDP for the three different region configurations and various values of ψ and υ.

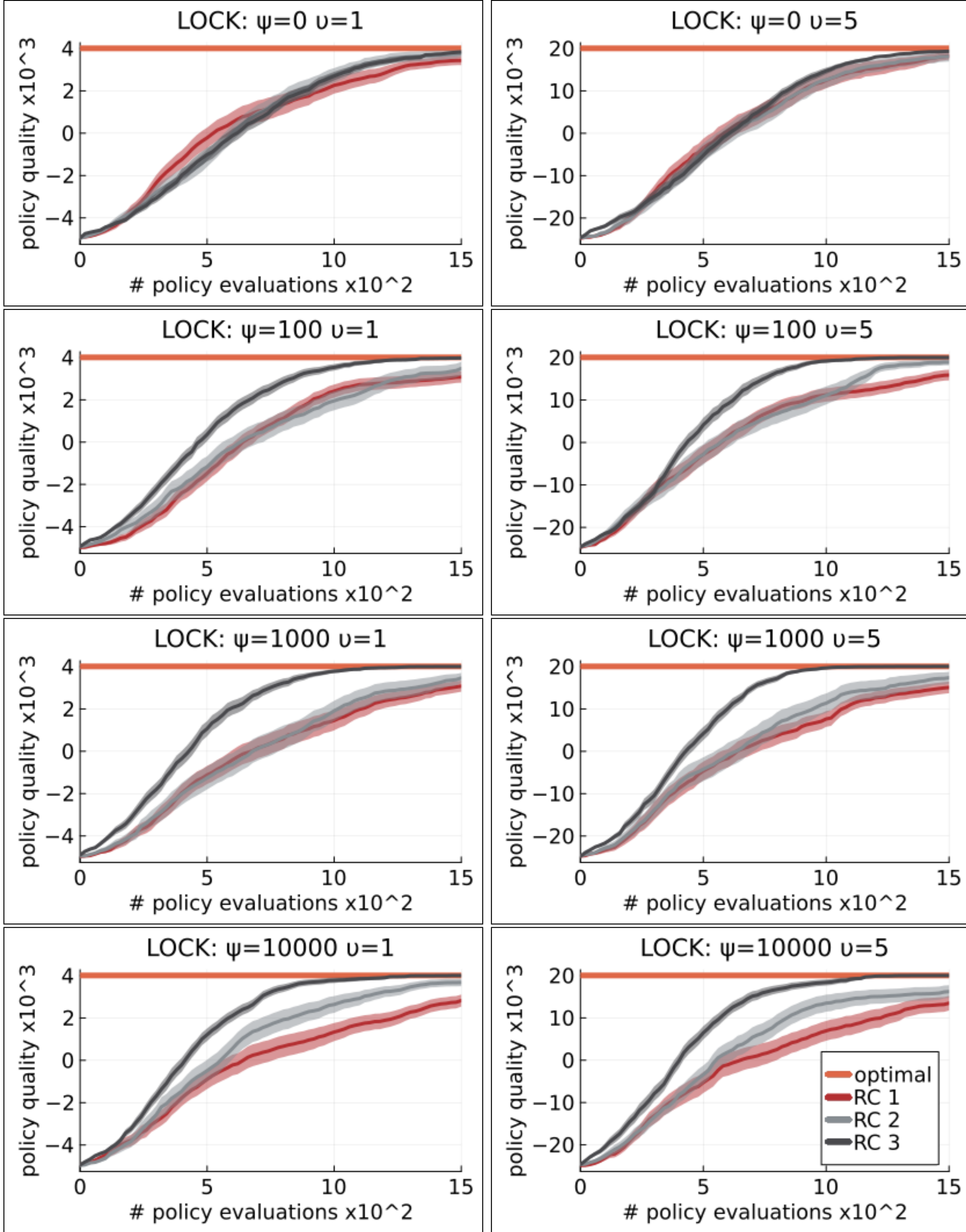

Figure B.3: Progress of the mean policy quality $(\upsilon \sum_{s \in \mathcal{S}} V^{\pi}(s))$ over the number of policy evaluations for the `LOCK` MDP for the three different region configurations and various values of ψ and υ.

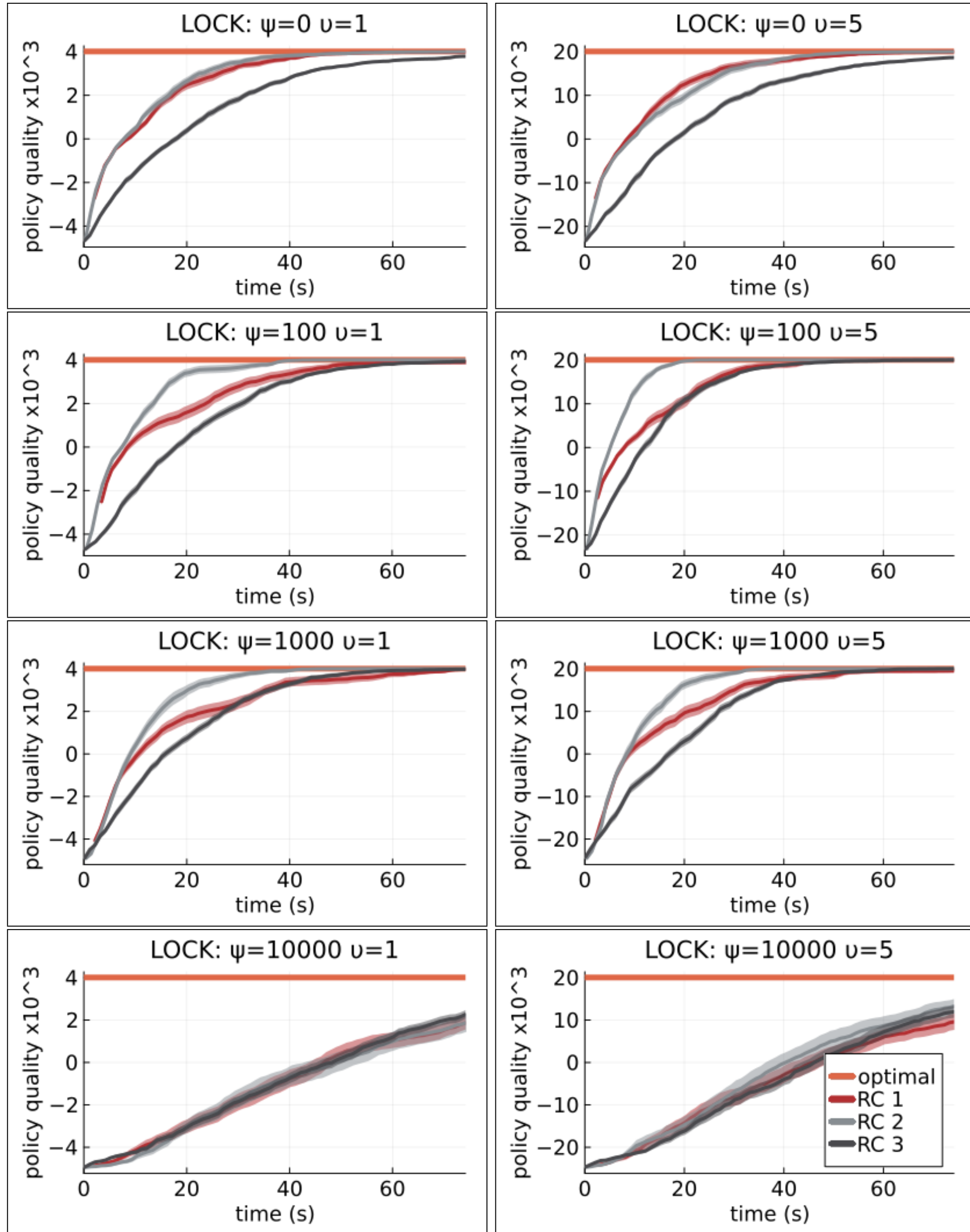

Figure B.4: Progress of the mean policy quality $(\upsilon \sum_{s \in \mathcal{S}} V^{\pi}(s))$ over the wall-clock time for the `LOCK` MDP for the three different region configurations and various values of ψ and υ.

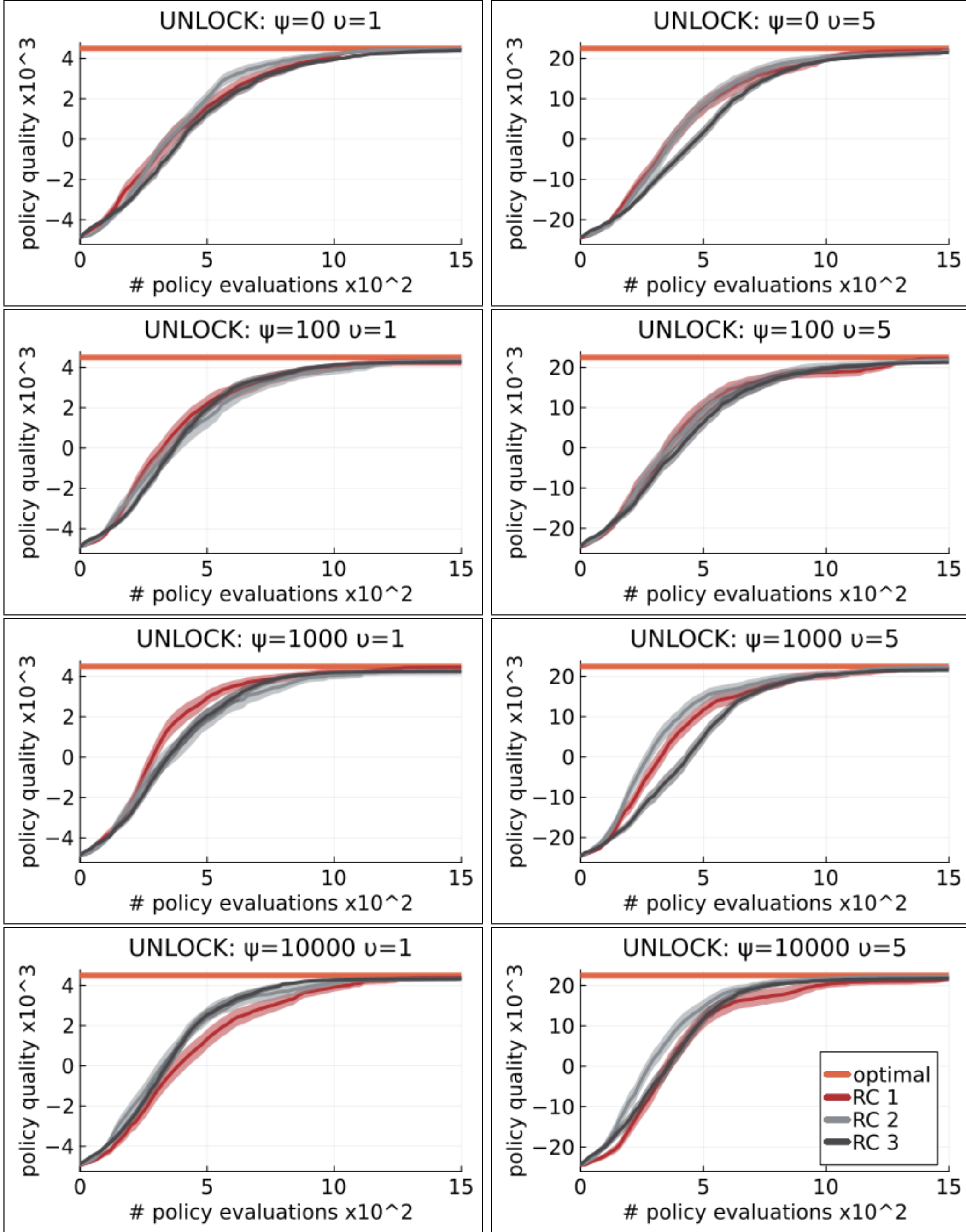

Figure B.5: Progress of the mean policy quality $(\upsilon \sum_{s \in \mathcal{S}} V^{\pi}(s))$ over the number of policy evaluations for the `UNLOCK` MDP for the three different region configurations and various values of ψ and υ.

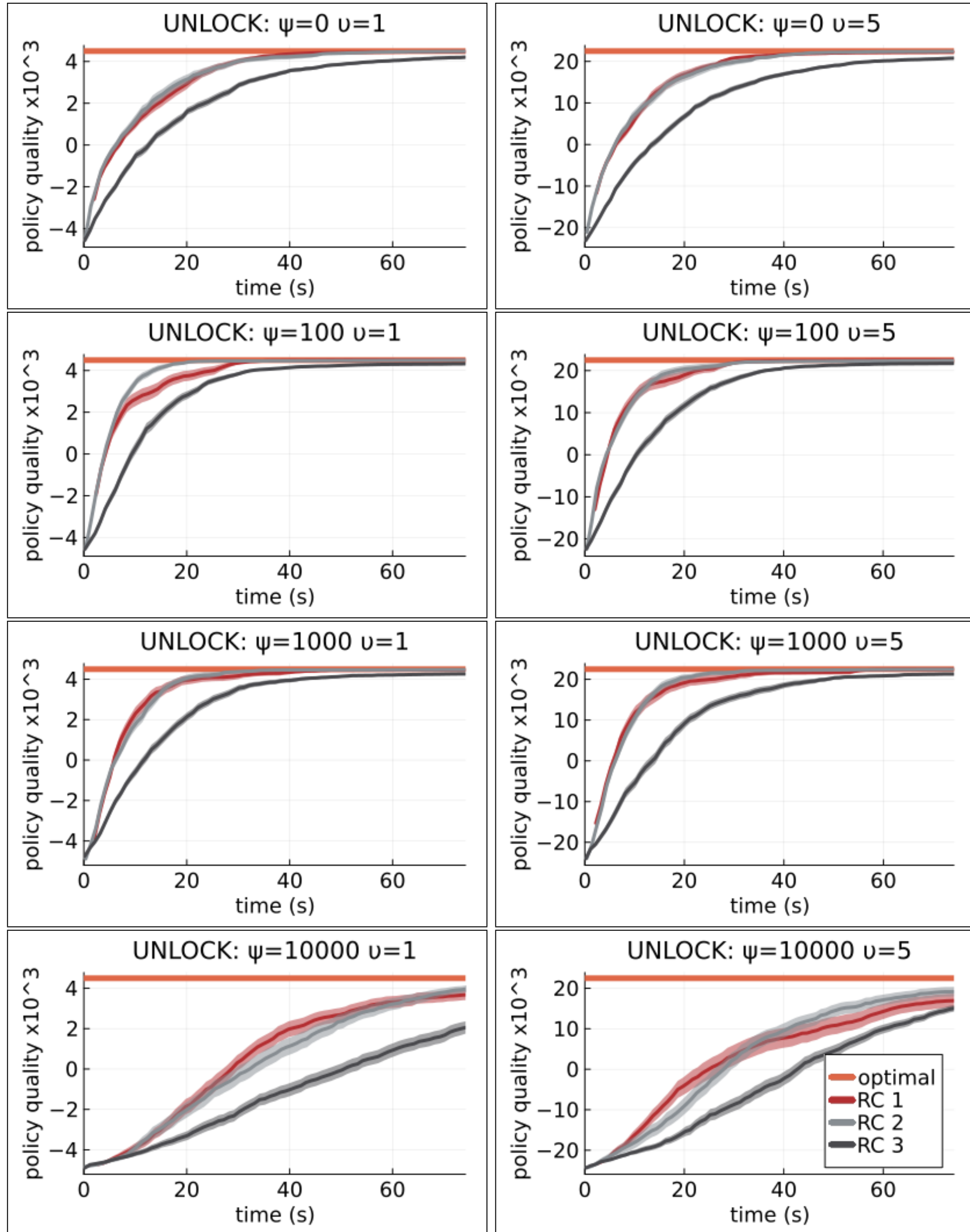

Figure B.6: Progress of the mean policy quality $(\upsilon \sum_{s \in \mathcal{S}} V^{\pi}(s))$ over the wall-clock time for the `UNLOCK` MDP for the three different region configurations and various values of ψ and υ.

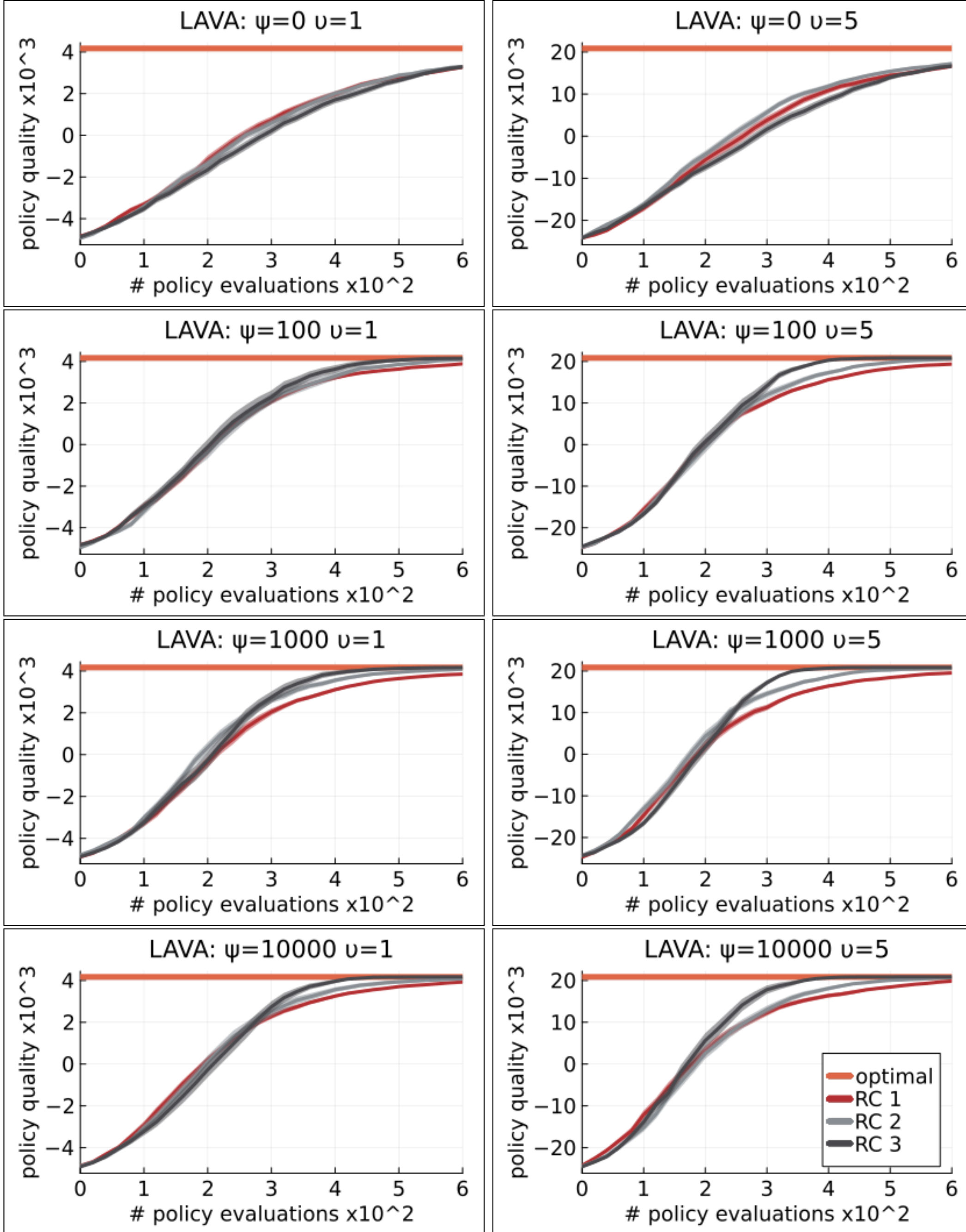

Figure B.7: Progress of the mean policy quality $(\upsilon \sum_{s \in \mathcal{S}} V^{\pi}(s))$ over the number of policy evaluations for the `LAVA` MDP for the three different region configurations and various values of ψ and υ.

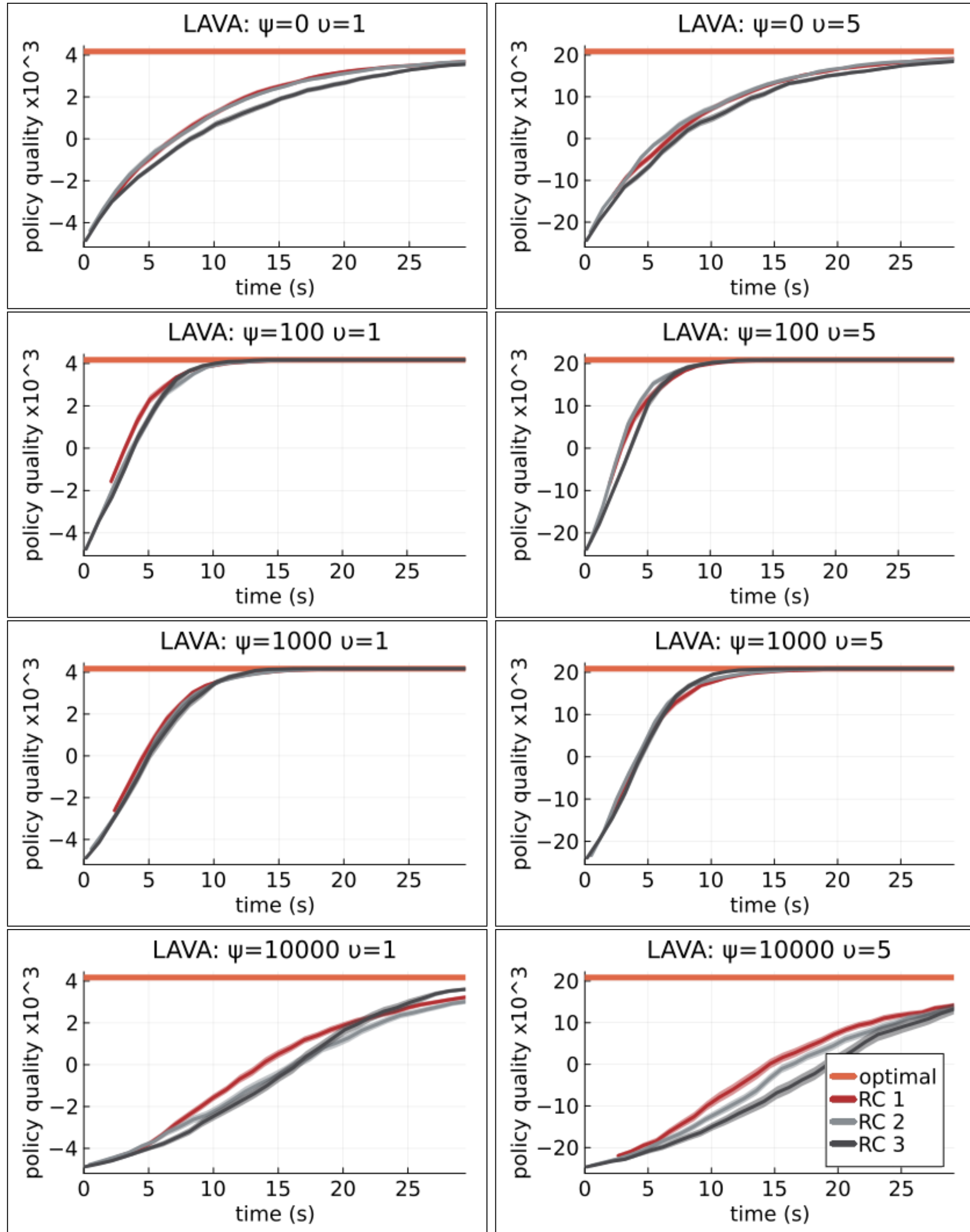

Figure B.8: Progress of the mean policy quality $(\upsilon \sum_{s \in \mathcal{S}} V^{\pi}(s))$ over the wall-clock time for the LAVA MDP for the three different region configurations and various values of ψ and υ.