Leveraging Prior Knowledge for Sample Efficient Reinforcement Learning

Ofir Marom

Supervised by Dr. Benjamin Rosman

A thesis presented for the degree of
Doctor of Philosophy

UNIVERSITY OF THE
WITWATERSRAND,
JOHANNESBURG

School of Computer Science and Applied Mathematics
University of the Witwatersrand
South Africa
March 2021

Abstract

A major goal in the field of Artificial Intelligence (AI) is the construction of autonomous agents that make effective decisions to complete real-world tasks. A promising mathematical framework to achieve this longstanding goal in AI is through the Reinforcement Learning (RL) paradigm. In RL an agent learns to maximise the long-term expected rewards it can receive from interacting with its environment. The agent achieves this by learning a policy that determines which optimal action to take in any given state that lead to such a maximal payoff.

For reasons of computational tractability, an RL task is typically described in terms of a Markov Decision Process (MDP). While there exist many algorithms with provable guarantees of convergence to an optimal policy given an MDP representation of an RL task, in practice there are numerous challenges in RL that limit its applicability for practical applications.

One of the keys to overcome these challenges is to improve the *sample efficiency* of RL algorithms. In RL, a sample refers to some action taken by the agent in its environment that leads the agent to gain some experience. The agent learns from this experience and, with enough samples, is able to learn an optimal policy for behaving in the environment. A sample efficient algorithm would extract maximum value from the smallest number of samples, thus improving the convergence rate to an optimal policy.

This thesis takes a step towards improving the sample efficiency of RL algorithms by leveraging *prior knowledge*. The term prior knowledge refers to any knowledge that the agent has about a task before starting to solve that task. Various techniques have demonstrated that injecting useful prior knowledge into RL algorithms can dramatically improve sample efficiency.

A natural question that arises with regards to prior knowledge is how such knowledge can be attained in the first place. While many techniques in RL treat prior knowledge as given, it is ideal if this knowledge can be learned. The research area of *transfer learning* provides a mechanism to learn prior knowledge. In the trans-

fer learning research problem, an agent attains prior knowledge from solving some source tasks that can then be applied to a related target task to improve learning performance.

The main contributions of this thesis are therefore as follows: firstly, we present a framework called Belief Reward Shaping (BRS) that can be incorporated with any model-free RL algorithm. BRS leverages prior knowledge about a task by augmenting the environment reward distribution with a prior distribution that encodes prior beliefs about useful sub-tasks to complete within a task. Provided this prior distribution encodes knowledge that is useful for solving the task, the agent can solve the task in a more sample efficient manner as the agent is guided by this prior knowledge to achieve specific sub-tasks within the task.

Secondly, we present an object-oriented formalism for RL called the Deictic Object-Oriented MDP (Deictic OO-MDP). Deictic OO-MDPs are based on the notion of a deictic predicate, which is a predicate that is grounded with respect to a single reference object that relates itself to lifted object classes. For certain domains, it is possible to use the Deictic OO-MDP formalism to efficiently learn a model of the transition dynamics of an environment that transfers across all tasks of a given domain. In addition, we extend the previously introduced Propositional OO-MDP formalism to learn a transferable model of the reward dynamics as well. This improves sample efficiently because the agent can reuse these models for any given task of the domain, rather than having to waste samples relearning it for each task.

Thirdly, we present an algorithm to efficiently learn likely-admissible heuristics from some source tasks of a domain. Once learned, the heuristic can be transferred to be used with a heuristic-based planning algorithm to produce likely-optimal plans for a new target task of the domain. Our approach utilises the notion of epistemic and aleatoric uncertainty to achieve this. We use epistemic uncertainty to efficiently explore task-space and generate source tasks that are of the right level to learn from. This approach to generating source tasks is more sample efficient than previous approaches that generated source tasks randomly. We further combine epistemic and aleatoric uncertainty to ensure that when the heuristic is used to solve source tasks during training, it plans with a value that is likely-admissible. This ensures that the final heuristic produces likely-optimal pans with low suboptimality.

Dedication

To my parents, Danny and Orly. And to my siblings, Dalit and Benny.

Declaration

I, Ofir Marom, declare that this Thesis is my own, unaided work. It is being submitted for the Degree of Doctor of Philosophy at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.

Signature

19/03/2021

Date

Acknowledgements

I imagine I am one of the scarcer cases of a PhD student as I started my PhD a bit later in life, in my early thirties. By that time, I had accumulated a reasonable amount of life experience. I believe this shaped my PhD journey, and gave me a capacity to appreciate certain things that I would have missed had I been younger.

I want to start by thanking my supervisor, Dr Benjamin Rosman. When I started my PhD, I felt uncertain of myself in this area. Doing novel research seemed like something above my capabilities. But a great mentor sees their student's potential, and the achievements I accomplished because you believed in my abilities has made me a stronger, more confident person today.

Dr Rosman has built a special lab where students are encouraged to pursue research that excites them. It has proven to be a formula for success and I am proud to have been a part of the lab's impressive growth. I want to give special thanks to Pravesh Ranchod, Steven James, Ritesh Ajoodha and Benjamin Van Niekerk. While the lab grew at a remarkable rate from the time I joined, initially it was a small group and you made me feel welcome when I was still finding my feet. I want to thank everyone else at the lab who made the sessions (and, just as importantly, the after-session lunches) fun and engaging.

To Aba and Ima. What words can I say to express my gratitude to you? When one is young, it is impossible to fully appreciate the sacrifices parents make to give their child the best chance for a good life. As I have grown older, I have reflected on what you did for me and the endless love you gave me. I hope if I ever have children of my own, I can be as good a parent to them as you were to me. To my sister Dalit, thanks for being someone I can speak to about things that are difficult for me to talk about with others. And to my brother Benny, it is a shame that we live far apart but we will always be connected because we are brothers.

Many others helped me during the course of my PhD. While I cannot list every person individually, I want to thank everyone who assisted me - be it with a dose of constructive criticism or an informal corridor chat about research direction.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Humans have a remarkable ability to quickly solve new tasks that they have never encountered before. This behaviour is evident right from early stages of infancy through to adulthood. A key component of this lies in the ability to effectively leverage prior knowledge from previous experiences that the human has had in its lifetime [Perkins and Salomon, 1992]. For example, a human infant learns to walk, which is then critical for the ability to run, which is then critical for the ability to play sports in later development.

Even the seemingly simplest of tasks that a human might encounter, such as walking from the dining room table to the kitchen to get a glass of water, become dauntingly complicated if the human would have to relearn everything from scratch to solve that task. However, because humans have the ability to effectively use prior knowledge, they do not need to gain much new experience, if any, to solve such tasks. Therefore, we can say that humans are highly efficient at solving such tasks because they do not need to gain much new experience to solve them.

A key goal in the field of Artificial Intelligence (AI) is the construction of autonomous agents that solve complex real-word tasks. Given the importance of prior knowledge for solving such tasks efficiently in humans, it makes sense that AI agents should have similar capabilities if they are to operate in the physical world with the same level of competence. In AI we can more formally define the sample efficiency of an automatous agents by a measure how many samples of data it needs to learn from in order to solve a task.

Unfortunately, imbuing AI agents with the ability to use prior knowledge to reach the same level of efficiency as humans has proven to be difficult. A further complication is that, in the same way that humans learn prior knowledge from past experience, such prior knowledge should be learned by an AI agent rather than being assumed

as known upfront. The research field of transfer learning provides a mechanism for an AI agent to learn prior knowledge. In the transfer learning research problem, an agent attains prior knowledge from solving some source tasks that can then be applied to a related target task to improve learning performance. While conceptually appealing, effective transfer learning in AI has proven to be difficult with many different approaches being proposed that only work in limited settings [Pan and Yang, 2010, Taylor and Stone, 2009]. Nevertheless, incorporating such knowledge into AI agents is critical to the field if it ever hopes to construct AI agents with the same level of competence at solving tasks as humans.

This thesis takes a step in the direction of improving the sample efficiency of AI agents by leveraging prior knowledge. In particular, we present three new frameworks under the Reinforcement Learning (RL) paradigm. RL is a sub-field of AI whereby an agent interacts with its environment with the aim of learning which actions to take in any given state to achieve an optimal payoff. The frameworks presented in this thesis build upon existing work in the areas of reward shaping, object-oriented representations and learning heuristic functions. Each framework presented in this thesis incorporates prior knowledge and can be shown to improve sample efficiency.

The rest of this chapter is organised as follows: in Section 1.1 we introduce the notion of sample efficiency and explain what it means for an agent to be sample efficient; in Section 1.2 we discuss prior knowledge and its importance in solving new tasks in a sample efficient manner; in Section 1.3 we discuss how prior knowledge can be attained through transfer learning; and we conclude with Section 1.5 where we discuss the structure for the rest of the thesis.

1.1 Sample Efficiency

Consider the task of getting up from a chair at a dining room table to get a glass of water from the kitchen, and suppose we asked a human to complete this task. For most humans this task is completely trivial to accomplish.

Now suppose we have a humanoid robot with all the necessary parts and joints to complete the same task. RL, a sub-field of AI, provides a formal mathematical framework for representing such a task for an autonomous agent to solve. However, given a mathematical representation of such a task under the RL paradigm, can the robot complete it as easily as the human?

The answer to this question depends largely on what prior knowledge the robot has at its disposal. If the robot has no prior knowledge, this seemingly simple

task becomes daunting. The robot must learn to: get up from the table; identify where the kitchen is; walk to the kitchen; get a glass from the cupboard; open the refrigerator; take out the water jug; pour water from the jug to the glass; put the water jug back in the refrigerator; close the refrigerator; walk back to the dining room table with the glass; and sit back on the chair.

If the robot has no prior knowledge, each one of these sub-tasks is itself dauntingly complicated. For example, walking to the kitchen requires the robot to learn how to move specific joints in its legs to achieve a walking motion. In such a case, we can say that the human is much more sample efficient than the humanoid robot at solving this task.

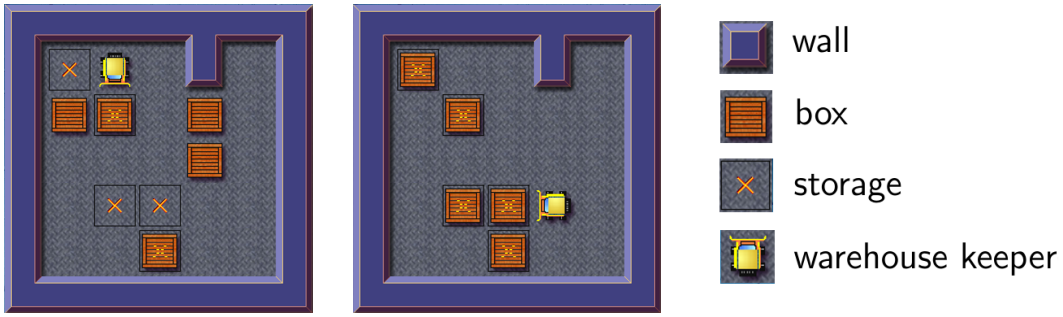
In RL when we talk about a sample, we refer to some experience that the agent gains from interacting with its environment. Sample efficiency therefore refers to the ability of an agent to learn how to solve a task in such a way so as to attain maximum knowledge from as few samples as possible. Therefore, a sample efficient robot would learn how walk to the kitchen with as few samples as possible, learning exactly which joints to move in its legs in a systematic way. Meanwhile, a sample inefficient robot would still learn how to do this but would waste a lot of samples in doing so, possibly taking many random actions that have nothing to do with walking such as flailing its arms or turning its head.

AI has made great strides in recent years on some very complex tasks such as backgammon, a suite of Atari 2600 games, Go, chess and shogi [Mnih et al., 2013,Silver et al., 2016,Silver et al., 2017,Tesauro, 1995,Tesauro, 2004]. AI agents that learn to play these games often outperform human experts. However, despite these impressive achievements these AI agents are highly sample inefficient compared to humans. As an example of this sample inefficiency, TD-Gammon produced an agent that achieved master-level performance in backgammon, but required 1,500,000 games to do so [Tesauro, 2004]. While stronger than most human players, a human player would need to play roughly 50 games every day for 80 years to experience the same number of games as TD-Gammon.

As we aim to scale AI to even larger problems, it is not conceivable that learning can be achieved in such a sample inefficient manner that relies on raw computation power. This makes sample efficiency in AI vitally important.

1.2 Prior Knowledge

Prior knowledge is key to the human ability to solve new tasks in a sample efficient manner. As an illustrative domain [1], consider the popular game Sokoban.[2] In Sokoban a warehouse keeper is tasked to push boxes to storage locations in a warehouse, but cannot do so if a box is against a wall or in front of another box. The initial game state and a final game state of the first level of the 'SokEvo' pack are shown in Figure 1.1.



(a) Initial game state. (b) A final game state.

Figure 1.1: Game states from the first level of the 'SokEvo' level pack.

Consider a human player who plays the game for the first time. If the player comes into the game without any prior knowledge, then the player would not know that they control the warehouse keeper by pressing the direction arrows on the controller. They would also not know that walls are immovable objects in the game, while boxes are movable. However, human players know these things before they start playing the game because they leverage prior knowledge. Such prior knowledge my come from past experience of playing similar video games or from their interactions with objects in the physical world. As a result, they start the game with an understanding of how the game operates and this aids them to complete the game, even before they have obtained any experience in the game itself.

Previous research has quantified the importance of prior knowledge in video games for human players and showed that by masking important visual cues that human players use as priors drastically reduces their gameplay performance [Dubey et al., 2018]. For example, if we were to change box sprites in Sokoban to be spider sprites, it would be unsurprising to find that humans would have a prior belief that they should avoid the spiders, even though the game mechanics have not changed in any way.

[1] In this thesis we refer to a domain as a group of related tasks. Therefore, the Sokoban domain comprises of all Sokoban tasks (or levels).

[2] All Sokoban images in this thesis are taken from JSoko: https://www.sokoban-online.de/jsoko/credits.

If an AI agent does not start a new task with such prior knowledge, it is unsurprising that they are less sample efficient at solving the task than a human player.

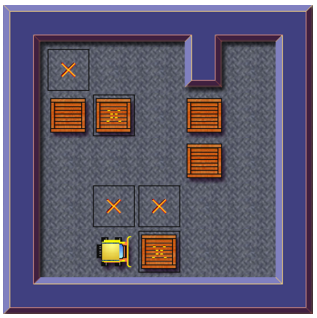
1.3 Transfer Learning

A key question with regards to prior knowledge is where such knowledge comes from in the first place. In humans, much prior knowledge comes from past experience that they then transfer to new tasks they encounter. For example, a human player that has never played Sokoban before may have played other video games, and knows from these games that they most likely control the sprite that looks like a tractor. They may also know from past experience in video games, as well as the physical world, that there are certain classes of immovable objects, like walls, and certain classes of movable objects, like boxes. Then it only takes them a few samples to confirm that in Sokoban wall and box objects operate in accordance with this segmentation.

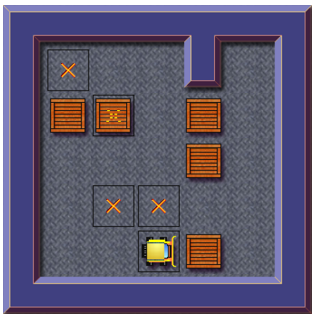
Human players are also able to infer the causal structure of new tasks in a transferable way. For example, consider Figure 1.2. A human player may suspect that they should interact with the boxes in the game, but may not be sure exactly how the game mechanics operate. Through experimentation, suppose a human player finds themselves in the game state as in Figure 1.2a, where the warehouse keeper has a box to its right and that box has an empty space to its right. The human player may try to press the 'right arrow' key on the controller and observe that the box moves to the empty space location, while the human moves to the box's previous location as shown in Figure 1.2b.

Thereafter, the human player would be able to transfer this knowledge to every new box they encountered in the game. For example, if the player found themselves in a game state as in Figure 1.2c then the player would know the effect of pressing the 'right arrow' key as shown in Figure 1.2d based on their prior experience without having to actually press the button.

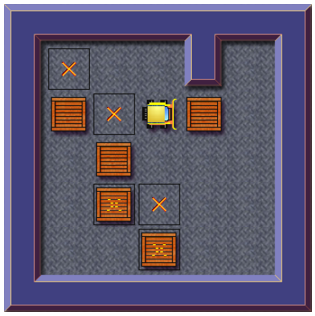
If we could incorporate the level of transfer that humans possess into AI agents, we could greatly improve their sample efficiency. In the field of AI, transfer learning refers to the research problem whereby an AI agent aims to gain knowledge by solving some source tasks and then use that knowledge to accelerate performance in a related target task. In particular with RL, much research has focused on finding techniques to incorporate transfer learning into existing RL algorithms [Taylor and Stone, 2009]. While many techniques have been proposed and have been shown to empirically improve sample efficiency, the transfer learning research problem has proven to be a difficult one. It is often unclear what to transfer, when to transfer or how to transfer effectively. Furthermore, transfer learning techniques tend to make



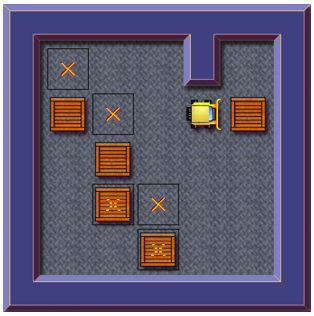
(a) Game state where the warehouse keeper can push the box adjacent to them.



(b) Game state after pressing 'right arrow' key in the game state as in Figure 1.2a.



(c) Having observed transition in game states from in Figures 1.2a to 1.2b, human players can transfer this knowledge to this new game state.



(d) Game state after pressing 'right arrow' key in the game state as in Figure 1.2c.

Figure 1.2: Example of transfer in Sokoban.

strong assumptions and work in limited settings.

Nevertheless, the transfer learning research problem remains an important research area in the field of AI as it provides a mechanism for an AI agent to attain prior knowledge that can then be used to improve sample efficiency when solving new tasks.

1.4 A Learning Framework for Autonomous Agents

Humans, as well as other organisms, learn much of their knowledge through interaction with their environment. For example, during infancy human babies show a tendency to experiment during play. They might pick up objects and drop them or push them over to see how they behave. During this process, the baby is learning about the causal structure of different object types in the world. It is easy to find numerous more examples of humans learning though interaction with their environment. Learning through direct interaction with the environment is a critical

component of cognitive development in living organisms.

Given the importance of learning through interaction in organisms, it is natural to want to develop a mathematical framework to model this process that can be applied to autonomous agents. In the late 1980s ideas from learning through trial and error in psychology of animal learning were combined with ideas from optimal control theory to produce the Reinforcement Learning (RL) paradigm [Sutton and Barto, 2018], a sub-field of Artificial Intelligence (AI).

At the core of RL is an agent and its environment. The goal of the agent is to interact with its environment to learn which sequence of actions to take in order to achieve an optimal payoff. In RL this sequence of actions is called the agent's *policy*. Optionally, an RL agent may also learn a model of the environment dynamics during its interactions. Such a model can then aid the agent in finding an optimal policy by giving the agent the ability to plan in its environment.

RL can therefore benefit from prior knowledge in all three phases of learning:

- when the agent is interacting with its environment to learn an optimal policy;
- when the agent is interacting with its environment to learn a model of the environment dynamics; and
- when the agent is planning with a model of its environment.

Unlike supervised learning, in RL the agent must learn without the aid of a teacher. This makes the RL framework highly appealing as the process of collecting labelled data for supervised learning tasks is often time consuming, expensive and error prone. RL has had many success stories in recent years and has produced master-level performance on a variety of benchmark games such as backgammon, a suite of Atari 2600 games, Go, chess and shogi [Tesauro, 1995, Tesauro, 2004, Mnih et al., 2013, Silver et al., 2016, Silver et al., 2017].

1.5 Thesis Structure

So far in this thesis we have discussed the importance of sample efficiency in AI agents to solve practical real-world tasks. Through illustration with Sokoban, we have demonstrated that prior knowledge is a powerful way to improve sample efficiency in AI agents while transfer learning provides a mechanism for AI agents to learn such prior knowledge.

In the rest of this thesis we present three novel frameworks within the RL paradigm that leverage prior knowledge, and demonstrate that each of these methods improves sample efficiently. In Chapter 2 we present an overview of RL that is required for

all three frameworks as well as a summary of the thesis contributions; in Chapter 3 we present a reward shaping framework called Belief Reward Shaping; in Chapter 4 we present a framework for zero-shot transfer with object-oriented representations; in Chapter 5 we present a framework to efficiently learn likely-admissible heuristics for planning; and in Chapter 6 we conclude with directions for future research and final remarks.

While the three contributions of this thesis are introduced independently, each can be categorised into to a different phase of learning in RL as discussed in Section 1.4. In particular:

- Chapter 3 leverages prior knowledge to learn an optimal policy;
- Chapter 4 leverages prior knowledge to learn a model of the environment dynamics; and
- Chapter 5 leverages prior knowledge for planning under a known model.

Chapter 2

Overview

2.1 Reinforcement Learning

In this section we outline the important components of Reinforcement Learning (RL) as needed for this thesis. For an in-depth introduction to RL see Sutton and Barto [Sutton and Barto, 2018].

The RL framework can be visually illustrated as per Figure 2.1. Under this setting, an agent interacts with its environment over a period of discrete time steps $t = 0, 1, 2, ...$ At each timestep t the agent observes some representation of the state of the environment $S_t \in \mathcal{S}$ where $\mathcal{S}$ is called the state-space. On the basis of this observation, the agent chooses to take some action $A_t \in \mathcal{A}(S_t)$ where $\mathcal{A}$ is called the action-space and $\mathcal{A}(S_t)$ are the actions available to the agent in state S_t. Thereafter, partly based the chosen action A_t, the agent finds itself in some new state $S_{t+1} \in \mathcal{S}$ and the agent receives some numerical reward $R_{t+1} \in \mathbb{R}$.

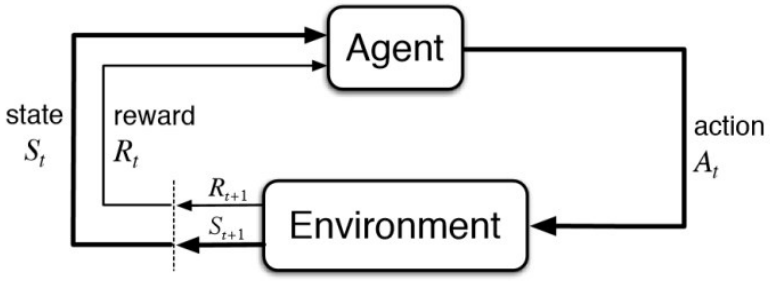


Figure 2.1: Agent-environment interaction in RL (image source: [Sutton and Barto, 2018]).

2.1.1 Markov Decision Process

For reasons of computational tractability, an RL task is typically formulated as a discrete-time, finite-state and finite-action Markov Decision Process (MDP). Such an MDP is defined by a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where:

- $\mathcal{S}$ is a finite set of states called the state-space;
- $\mathcal{A}$ is a finite set of all actions called the action-space, while $\mathcal{A}(s)$ is the actions available in state $s \in \mathcal{S}$;
- $\mathcal{P}$ is the transition dynamics for which $\mathcal{P}(s'|s, a)$ returns the transition probability for transitioning to state $s' \in \mathcal{S}$ conditional on the agent being in state s and taking action $a \in \mathcal{A}(s)$;
- $\mathcal{R}$ is the reward dynamics for which $\mathcal{R}(s, a, s')$ returns the (stationary, finite variance) distribution that governs the reward signal that the agent receives when in state s, takes action a and transitions to state s'; and
- $\gamma \in [0, 1)$ is a discount rate that controls the trade-off between the importance of immediate and future rewards.

An additional component of an MDP is $\rho(s_0)$ for $s_0 \in \mathcal{S}$ that is a distribution over possible start states for the MDP. For notational simplicity, this component is typically excluded from the MDP definition in the literature, but is implicit in the definition of an MDP. We adopt this convention in this thesis.

An RL task represented as an MDP has the *Markov property*. The Markov property is a property of a stochastic process whereby the probability distribution of the future state of the process depends only on the present state. Consequently, a stochastic process that has the Markov property is said to be 'memoryless'. Making the assumption that a process follows the Markov property can often lead to tractable solutions. The Markov property exists in MDPs because the transition dynamics and reward dynamics do not depend on any information prior to the current state and action observed.

In RL the agent has no control over (and often does not even know) the environment dynamics, $\mathcal{P}$ and $\mathcal{R}$. What the agent does control is the policy it follow in the environment. Formally, a policy is defined by $\pi(a|s)$ and represents the probability of taking action a conditional on being in state s. The goal of an RL agent is to find an optimal policy π_* that maximises the expected future discounted rewards. More formally, we first define the return of a process as the sum of discounted rewards:

$$G_t = \sum_{k=t+1}^{\infty} \gamma^{k-t-1} R_k \tag{2.1}$$

Then the value function under a fixed policy π is defined as:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s \right] \\ &= \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|a, s)(r(s, a, s') + \gamma V_\pi(s')), \end{aligned} \tag{2.2}$$

where $r(s, a, s') = \mathbb{E}\left[\mathcal{R}(s, a, s')\right]$. The value function represents the expected return that the agent receives if the agent is currently in state s and follows the policy π. Equation 2.2 is called the Bellman equation for V_π. The interpretation of the equation is as follows: given a state s, the value of that state is the sum of the immediate expected reward and the value of the next state under the policy π. Since the policy and transition dynamics are stochastic, we take expectations with respect to possible actions and next states.

Similar to the value functions, we can define the action-value function for a fixed policy π as:

$$\begin{aligned} Q_\pi(s, a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &= \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a)(r(s, a, s') + \gamma V_\pi(s')). \end{aligned} \tag{2.3}$$

The action-value function is similar to the value function, but we now condition on the immediate action that the agent takes as well. Therefore, an action-value function represents the expected return that the agent receives for following the policy π given that the agent is currently in state s and takes action a.

Given the definition of the action-value function, we can now define the optimal action-value function as the action-value function that maximises the expected return of for all possible states and actions. That is:

$$Q_*(s, a) = \sup_\pi Q_\pi(s, a) \tag{2.4}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

Since

$$V_*(s) = \max_{a \in \mathcal{A}(s)} Q_*(s, a), \tag{2.5}$$

We can further express Q_* in terms of the Bellman equation as follows:

$$Q_*(s, a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a)(r(s, a, s') + \gamma \max_{a' \in \mathcal{A}(s')} Q_*(s', a')). \tag{2.6}$$

Given Q_*, an optimal policy π_* can be derived for any given state s by choosing the action b such that

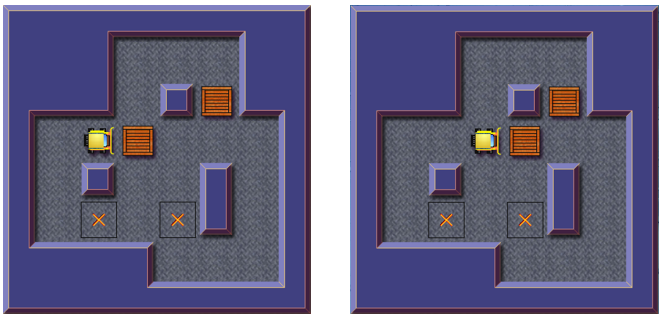
$$b = \arg\max_{a \in \mathcal{A}(s)} Q_*(s, a). \tag{2.7}$$

So far, we have defined RL tasks in terms of continuing tasks with an infinite time horizons. However, in many cases we want to model tasks that end once some terminal (or goal) state has been reached by the agent. For example, if we were using RL to solve a level of Sokoban, a terminal state is any state where all boxes are at storage locations. This is called an episodic task which has a finite time horizon. Episodic tasks can be incorporated into the RL framework by assuming that once the agent reaches a terminal state, the agent enters into an absorbing state that always transitions back to itself and always gives a reward of zero. Episodic tasks also have the property that they can allow $\gamma = 1$ because the task terminates, and so the sum of undiscounted rewards is finite. In this thesis we restrict attention to episodic tasks.

In some tasks, the transition dynamics or reward dynamics are said to be deterministic. If $\mathcal{P}$ is deterministic then for all possible (s, a) passed to $\mathcal{P}$ there is one possible next state s' that occurs with probability one. Therefore we may write $\mathcal{P}(s'|s, a) = \mathbb{1}(s = s')$ where $\mathbb{1}$ is the indicator function. If $\mathcal{R}$ is deterministic, then for all possible (s, a, s') passed to $\mathcal{R}$, a deterministic scalar is returned. Therefore we may write $\mathcal{R}(s, a, s') = r(s, a, s')$. If both $\mathcal{P}$ and $\mathcal{R}$ are deterministic we say that the environment has deterministic dynamics or, more concisely, that the environment is deterministic.

2.1.2 Sokoban as an MDP

To make the MDP formalisation more concrete let us give an example of how the Sokoban level as in Figure 2.2 could be described as an MDP.



(a) Initial game state.

(b) Game state after pressing 'left arrow' key from game state in Figure 2.2a.

Figure 2.2: Game states from level ten of the 'Micro-Cosmos' Level Pack.

The action-space is $\mathcal{A} = \{North, East, South, East\}$ as these are the actions available to the agent in the game and move the warehouse keeper in the respective directions in the warehouse.

There are potentially multiple ways to define the state-space for this MDP. For simplicity, let us consider a state-space representation where each state keeps track of the (x, y) coordinates of the warehouse keeper and the boxes in the warehouse. That is, each state is defined by $s = (p_x, p_y, b_x^1, b_y^1, b_x^2, b_y^2)$ where p_x and p_y are the x and y coordinates of the warehouse keeper, and the same format applies to the two boxes b^1 and b^2. Since the wall and storage locations cannot change, we do not need keep track of them for the purposes of solving this task. Then the game state as in Figure 2.2a can be represented by $s = (2, 4, 3, 4, 5, 5)$. The state-space $\mathcal{S}$ contains all possible configurations of states.

The terminal states in Sokoban are the states where all boxes are at storage locations, so for this task the set of terminal states is $\mathcal{S}_{terminal} = \{s \mid s \in \mathcal{S} \text{ and } (s = (., ., 2, 2, 4, 2) \text{ or } s = (., ., 4, 2, 2, 2))\}$.

Since the agent can take the same set of actions in each state, we have that for this task $\mathcal{A}(s) = \mathcal{A}$ for all states $s \in \mathcal{S}$.

The transition dynamics encode the game mechanics and, in the case of Sokoban, are deterministic. For example, suppose the agent takes the action *East* from state $s = (2, 4, 3, 4, 5, 5)$. Then the next state $s' = (3, 4, 4, 4, 2, 5)$. So $\mathcal{P}(s, East) = \mathbb{1}(s = s')$.

The reward dynamics in Sokoban must specify what we want the agent to achieve. We want the agent to reach a terminal state in as few steps as possible. A common way of specifying the reward dynamics in episodic tasks is to give the agent a small deterministic negative reward for each step taken, say -1, except for steps that lead to a terminal state that have a high positive deterministic reward, say 100.

This definition encourages the agent to reach a terminal state in as few steps as possible because the agent incurs a negative reward for each wasted step taken and is incentivised to reach a terminal state to get the large positive reward.

Since this is an episodic task, we can set $\gamma = 1$ and since this task has only a single possible start state, $\rho(s_0) = \mathbb{1}(s_0 = (2, 4, 3, 4, 5, 5))$.

There are various policies available to the agent. One possible policy is a deterministic policy that always chooses the action *East* for any state. Another possible policy is a stochastic policy that chooses a random action for any state. However, neither of these policies is optimal.

Given such an MDP we now want to solve the MDP by finding an optimal policy that maximises the return that the agent receives. In this thesis we are interested in three broad approaches to learn an optimal value (or action-value) function, from which an optimal policy can then be derived: the planning approach, which is discussed in Section 2.1.3; the model-free approach, which is discussed in Section 2.1.4; and the model-based approach, which is discussed in Section 2.1.4.[1]

2.1.3 Planning under Known Dynamics

In the case where $\mathcal{P}$ and $\mathcal{R}$ are known by the agent, an RL task simplifies to a planning task. With planning, the agent can learn an optimal policy without needing to gain experience by interacting with its environment. In RL planning is often done with Dynamic Programming (DP). The term DP that refers to a suite of algorithms that leveraging the recursive nature of the Bellman equation to plan. One of the most well-known DP algorithms in RL to learn π_* is called value iteration [Bertsekas, 1987]. Value iteration works by iterating over the state-space and action-space of an MDP and applying the Bellman update for the optimal action-value function as per equation 2.6 until convergence. Algorithm 1 outlines this procedure.

2.1.4 Model-Free Reinforcement Learning

If $\mathcal{P}$ and $\mathcal{R}$ are unknown, the agent must learn π_* through interacting with its environment and gaining experience. In model-free RL the agent does not aim to learn $\mathcal{P}$ and $\mathcal{R}$ and goes straight from experience to a policy.

One of the most well-known model-free algorithms in RL is Q-learning [Watkins, 1989]. At the core of the Q-learning algorithm is the update rule

[1] An alternative approach not considered in this thesis is the policy-based approach that directly learns an optimal policy from experience.

Algorithm 1: *Value Iteration*: the value iteration algorithm for episodic tasks.

Input: $\theta > 0$ `// small constant controlling accuracy to optimal action-value function`

initialise $Q(s, a)$ arbitrarily except for $Q(\mathcal{S}_{terminal}, .) = 0$.
repeat
  $\Delta \leftarrow 0$
  foreach $s \in \mathcal{S}$ **do**
    foreach $a \in \mathcal{A}(s)$ **do**
      $q \leftarrow Q(s, a)$
      $Q(s, a) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a)(r(s, a, s') + \gamma \max_{a' \in \mathcal{A}(s')} Q(s', a'))$
      $\Delta \leftarrow \max(\Delta, |q - Q(s, a)|)$
    end
  end
until $\Delta < \theta$
return Q

$$Q(s, a) \leftarrow Q(s, a) + \alpha(s, a)(r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s', a') - Q(s, a)), \tag{2.8}$$

where $\alpha(s, a) \in (0, 1)$ is a learning rate parameter for state s and action a. The term $r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s', a')$ is called the target as it is an improved estimate of $Q_*(s, a)$ over of the current $Q(s, a)$. It can be proved that, under certain conditions, applying this update rule over timesteps leads to convergence to Q_*. In the RL literature, the technique that Q-learning uses to apply such updates is called one-step temporal-difference (TD) because the target is constructed by looking one step ahead of the current timestep to see what the next reward is.

Initial proofs of Q-learning convergence to an optimal policy for finite MDPs required that

$$\sum_{t=0}^{\infty} \alpha_t(s, a) = \infty \quad \text{and} \quad \sum_{t=0}^{\infty} \alpha_t^2(s, a) < \infty, \tag{2.9}$$

where α_t indexes $\alpha(s, a)$ at timestep t [Jaakkola et al., 1994]. These criteria ensure that all state-action pairs are visited infinitely often.

In practice, setting $\alpha_t(s, a)$ to ensure the criteria in equation 2.9 for an MDP can be difficult. Fortunately, Q-learning convergence is guaranteed so long as all state-action pairs are visited infinitely often. An alternative way to achieve this is to use a constant value for $\alpha \in (0, 1)$ with an ϵ-greedy exploration policy. Such a policy takes random actions with probability $\epsilon \in (0, 1)$ and with probability $1-\epsilon$ takes an optimal action from the policy derived from the current action-value function maintained in the algorithm - that is by selecting action $b = \arg\max_{a \in A(s)} Q(s, a)$ when in state

s. In practice, Q-learning with ϵ-greedy exploration is easy to implement and is therefore widely used. Q-learning with ϵ-greedy exploration is outlined in Algorithm 2.

Algorithm 2: *Q-Learning*: the Q-learning algorithm for episodic tasks with ϵ-greedy exploration.

Input: α, ϵ

1 initialise $Q(s, a)$ arbitrarily except for $Q(\mathcal{S}_{terminal}, .) = 0$.
2 **repeat**
3 start episode at initial state s
4 **while** $s \notin \mathcal{S}_{terminal}$ **do**
5 choose action $a \in \mathcal{A}(s)$ derived from ϵ-greedy policy
6 observe next state s' and reward r
7 $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s', a') - Q(s, a))$
8 $s \leftarrow s'$
9 **end**
10 **until** *convergence*
11 return Q

2.1.5 Model-Based Reinforcement Learning

In model-based RL the agent does not initially know $\mathcal{P}$ or $\mathcal{R}$ but learns a model of them through experience. We denote the model of $\mathcal{P}$ with $\mathcal{P}_{model}$ and the model of $\mathcal{R}$ with $\mathcal{R}_{model}$. These models can then aid the agent in learning π_* by simulating experience. A conceptually simple model-based RL algorithm is called Dyna-Q [Sutton, 1990] which is outlined in Algorithm 3. For simplicity, the algorithm is presented for deterministic environments.

The algorithm is similar to Q-learning but the agent now learns models of $\mathcal{P}$ and $\mathcal{R}$ while interacting with its environment. In line 12 a loop is initialised that applies N one-step TD updates from simulated experience generated from the models. These additional updates can accelerate convergence to an optimal policy.

Dyna-Q leverages one-step TD updates from simulated experience to learn π_*. An alternative approach in model-based RL is to use an algorithm like R_{max} [Brafman and Tennenholtz, 2002] whereby the agent iteratively learns more accurate models of $\mathcal{P}$ and $\mathcal{R}$ by running an exact planning algorithm like value iteration to learn an optimal policy under an incorrect model that is constructed in such a way so as to drive the agent to explore unknown parts of the environment. The R_{max} algorithm is described in greater detail in Chapter 4.

Algorithm 3: *Dyna Q- Learning*: the Dyna-Q algorithm for episodic tasks, ϵ-greedy exploration and deterministic environments.

Input: α, ϵ, N `// number of simulated experiences to apply per timestep`

initialise $Q(s,a)$ arbitrarily except for $Q(\mathcal{S}_{terminal},.) = 0$.
initialise $\mathcal{P}_{model}$ and $\mathcal{R}_{model}$
repeat
  start episode at initial state s
  while $s \notin \mathcal{S}_{terminal}$ **do**
    choose action $a \in \mathcal{A}(s)$ using ϵ-greedy policy
    observe next state s' and reward r
    $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s',a') - Q(s,a))$
    $\mathcal{P}_{model}(s,a) \leftarrow s'$
    $\mathcal{R}_{model}(s,a,s') \leftarrow r$
    $s \leftarrow s'$
    for $i \in [1:N]$ **do**
      sample s_i from prevously observed states
      sample a_i from previousy observed action in $\mathcal{A}(s_i)$
      $s'_i \leftarrow \mathcal{P}_{model}(s_i, a_i)$
      $r_i \leftarrow \mathcal{R}_{model}(s_i, a_i, s'_i)$
      $Q(s_i,a_i) \leftarrow Q(s_i,a_i) + \alpha(r_i + \gamma \max_{a' \in \mathcal{A}(s'_i)} Q(s'_i,a') - Q(s_i,a_i))$
    end
  end
until *convergence*
return Q

2.2 Challenges in Reinforcement Learning

The RL framework is very appealing because, in principle, it can be applied to solve a variety of important problems from learning to play games to controlling humanoid robots in the physical world. Furthermore, there are some fairly simple algorithms with convergence guarantees to optimal policies. Unfortunately, RL has some major challenges that need to be overcome before it can be more widely applicable.

In this section we discuss some of these challenges. In Section 2.2.1 we discuss the challenge of sparse rewards; in Section 2.2.2 we discuss the challenge of the 'curse of dimensionality'; and in Section 2.2.3 we discuss the challenge of non-transferable representations.

2.2.1 Sparse Rewards

RL suffers from the problem of sparse rewards. Sparse rewards are commonly defined as a reward structure where the agent receives little feedback on how they are performing at a task while learning to solve it [Hare, 2019, Gaina et al., 2019]. In episodic tasks, this commonly manifests as a reward structure where the agent receives mostly non-positive rewards with few positive rewards while interacting

with its environment. For example, in the Sokoban MDP described in Section 2.1.2 we use a reward signal that is -1 for all steps except for steps that reach a terminal state which produces a reward of 100. Therefore, the agent gets only one, highly delayed, positive reward to learn from when solving the task.[2]

To see why this is a problem, suppose we apply the Q-learning algorithm as in Algorithm 2 to solve the Sokoban MDP and suppose we use the common initialisation strategy of setting the action-value function to be zero for all state-action pairs. Since the algorithm uses ϵ-greedy exploration, the starting point for the algorithm is to take random steps. Since the agent gets a reward of -1 for each step, the action-value function will become more and more negative as it visits more non-terminal states. The agent must get lucky to take a sequence of random steps that lead to a terminal state to receive the positive reward. This positive reward is the only feedback that the agent gets throughout the task that it is performing the task as intended.

Such a sequence of random steps is plausible for an MDP with small state-space and action-space. But now consider an MDP for a much larger Sokoban level as shown in Figure 2.3. In this case the agent would have to get extremely lucky to take a sequence of random actions that get it to a terminal state. This is why, while Q-learning may have convergence guarantees to an optimal policy, in practice the rate of convergence is not adequate for any practical application with sparse rewards.

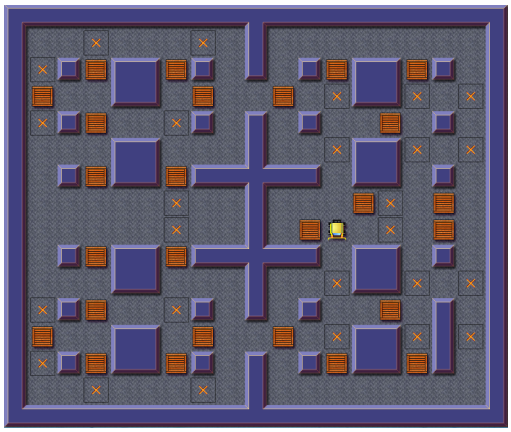


Figure 2.3: Initial game state from the first level of the 'Grigr 2001' level pack.

A seemingly obvious solution to this problem is to adjust the reward dynamics to be less sparse. Unfortunately, this can lead to unintended consequences where the optimal policy under the non-sparse reward dynamics does not produce an agent that behaves as intended. We illustrate this further in Chapter 3.

[2]Note that sparseness not related to the sign of the rewards. For example, in Sokoban we could redefine the rewards to be -1 for all steps (including steps that reach a terminal state). Such a reward structure is still sparse because it does not provide immediate informative feedback to the agent while solving the task.

2.2.2 Curse of Dimensionality

A further major issue in RL is known as the 'curse of dimensionality' where the size of the state-space of an MDP explodes exponentially as the number of state-space variables increase linearly. To see why this happens, consider the Sokoban MDP described in Section 2.1.2. Since this Sokoban level has only two boxes in a 6×6 warehouse, the size of the state-space is roughly $6^6 = 46,656$.[3] This number is certainly manageable.

Now suppose we want to solve the level as in Figure 2.3. This larger level has 26 boxes in a 17×14 size warehouse. Since we use two variables per box in our state description, this results in (including the warehouse keeper) 54 state variables. In total, the size of the state-space for this MDP is then roughly $17^{27} \times 14^{27} = 1.47088 \times 10^{64}$, which is enormous.

Applying any of the algorithms described in Section 2.1.1 is not feasible to this MDP. Even value iteration, which assumed known environment dynamics, cannot solve such an MDP in any meaningful amount of time. Recall that value iterating requires multiple sweeps over the state-space of an MDP to converge, and when the state-space is so large even one sweep becomes infeasible.

2.2.3 Non-Transferable Representations

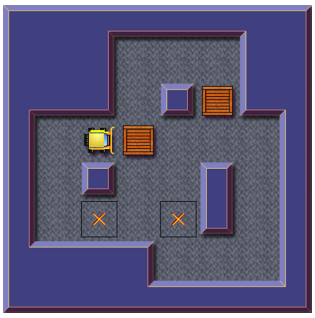
A further major challenge in RL has to do with non-transferable representations. In the same way that a human player would not start with a complicated level of Sokoban, but rather build up their experience and skills in the game from easier levels, so too should RL agents. In order to do this, RL agents should be able to leverage prior knowledge and transfer from previously solved Sokoban tasks to new Sokoban tasks.

Unfortunately, the intuitive representation for the Sokoban MDP as described in Section 2.1.2 does not lend itself to transfer. To see why, suppose we apply the Dyna-Q algorithm to learn Q_*, $\mathcal{P}_{model}$ and $\mathcal{R}_{model}$ for an MDP that describes level 2.4a.

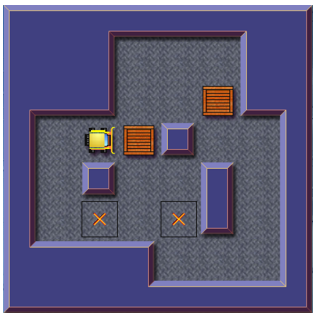
Given that we have these components, we would hope to be able to reuse them to accelerate solving a new MDP that describes a different Sokoban level. Consider the Sokoban level that is a slightly tweaked version of the first level as shown in Figure 2.4b. All that has changed from the first level is one wall location and one storage location. Unfortunately, even with such a minor change to the solved task,

[3] This is a crude upper bound. The actual size is smaller because we ignore wall locations, as well as the fact that in the game objects like boxes cannot be on top of each other. Furthermore, some states may not be reachable from the start state of the task.

none of the components learned in the source MDP transfer to the target MDP that describes this tweaked level.



(a) Sokoban level described in Section 2.1.2.



(b) A slightly tweaked Sokoban level from the one in Figure 2.4a.

Figure 2.4: Two similar Sokoban levels. Without transferable representations, it is not possible to reuse components between them.

For example, consider the state $s = (2, 4, 3, 4, 5, 5)$ with action *East*. In the source MDP this would result in $s' = (3, 4, 4, 4, 5, 5)$, but in the target MDP it results in $s' = (2, 4, 3, 4, 5, 5)$ because there is now a wall west of the box location adjacent to the warehouse keeper. Therefore, $\mathcal{P}_{model}$ learned in the source MDP does not transfer to this target MDP. The same kind of problem arises with Q_* and $\mathcal{R}_{model}$ - they do not transfer because the set of terminal states in the target MDP have changed.

2.3 Performance of Reinforcement Learning Algorithms

All the algorithms described in Section 2.1 guarantee convergence to an optimal policy. While such a guarantee initially seems reassuring, the challenges described in Section 2.2 imply that these algorithms do not perform well in practice. Therefore, convergence guarantees of RL algorithms are not sufficient. In addition, we also care about performance.

In particular, we would like our algorithms to converge as quickly as possible and furthermore, to use up as little computational resources as possible. There are three main categories for measuring the performance of RL algorithms: computational complexity as discussed in Section 2.3.1; space complexity as discussed in Section 2.3.2; and learning complexity as discussed in Section 2.3.3. While we introduce computational complexity and space complexity in this section for completeness, our main focus is on learning complexity.

2.3.1 Computational Complexity

In RL, an agent finds itself in some state and needs to choose an action to take, which results in the agent gaining some experience in the form of a next state and reward. There are two computations that need to be performed per step: 1) the agent must pick an action to take and; 2) the agent must learn from the experience gained from taking that action to update its policy. The computational complexity of both of these computations depends largely on the algorithm being used.

Consider the Q-learning algorithm described in Section 2.1.4. The action-taking component has computational complexity $\mathcal{O}(|\mathcal{A}(s)|)$ where $|.|$ represents the size of a set. This is because, given a state s, we need to find the action for which the action-value function has the highest value and this requires $|\mathcal{A}(s)|$ evaluations. Meanwhile, the learning-component of Q-learning comprises of making a single TD update per step which has $\mathcal{O}(1)$ computational complexity.

For comparison, R_{max} runs an exact planning algorithm, like value iteration, at each step to get an optimal policy under an incomplete model of the environment dynamics. Value iteration has a worst-case computational complexity $\mathcal{O}(|S|^2|A|)$ per sweep. Therefore, Q-learning has far lower per-step computational complexity than R_{max}. However, the number of steps required to solve a task with Q-learning may be higher, resulting in a higher per-task computational complexity.

2.3.2 Space Complexity

Space complexity measures the amount of space required by an algorithm to maintain its data structures. For example, a tabular action-value function that stores a value for each (s, a) has a space complexity of $\mathcal{O}(|S||A|)$. One way to get around this high space complexity is to use function approximators, such as linear methods or neural networks [Sutton and Barto, 2018], to represent the action-value function. Such functions use a set of parameters to compactly store the action-value function. Space complexity is not a focus of this thesis but we include it for completeness.

2.3.3 Learning Complexity

Learning complexity measures how much experience an agent needs to gain in its environment to learn an optimal policy. In RL, an agent must explore its environment and, in the process, take sub-optimal actions to gain experience before it can learn an optimal policy. The fewer sub-optimal actions the agent takes to learn an optimal policy the lower the learning complexity. While Q-learning with ϵ-greedy exploration may have low per-step computational complexity, it has high learning

complexity. This is because the agent starts out by taking random actions in the environment, rather than having a more systematic way of exploring.

Much research on RL has focused on developing algorithms with improved learning complexity [Jin et al., 2018, Brafman and Tennenholtz, 2002, Kearns and Singh, 1998, Strehl et al., 2006, Strehl and Littman, 2005, Kakade, 2003]. Many of these algorithms, such as R_{max} [Brafman and Tennenholtz, 2002], rely on the principle of *optimism in the face of uncertainty*. The idea behind this principle is that, when developing an exploration strategy, the agent should assume that reaching any part of the environment that it has yet to fully explore leads to the highest possible payoff. This encourages the agent to explore these areas, thus learning the truth about them and moving onto new, unknown, areas that the agent is still uncertain about. Algorithms that rely on this principle are often able to produce polynomial bounds on learning complexity.

Sample efficiency and learning complexity are inverse concepts: reducing learning complexity increases sample efficiency and vice versa. In this thesis we use the terms 'improve sample efficiency' and 'reduce learning complexity' interchangeably.

2.4 Improving Sample Efficiency with Prior Knowledge

One way to improve the sample efficiency of RL algorithms is with improved exploration strategies, such as using the optimism in the face of uncertainty principle. An alternative approach is to inject prior knowledge into existing RL algorithms. If this prior knowledge is well-specified it can improve sample efficiency as the agent can utilise this prior knowledge to learn an optimal policy with fewer samples.

As a simple example of how prior knowledge can be incorporated into RL to improve sample efficiency, consider the Q-learning algorithm with ϵ-greedy exploration described in Section 2.1.4. The Q-learning algorithms converges to Q_* no matter how the action-value function is initialised.

Suppose that we have some prior knowledge about which state-action pairs have high or low value. Then instead of arbitrary initialisation, we can initialise the action-value function with this prior knowledge. This can lead to improved sample efficiency because the algorithm uses the policy derived from the current action-value function with probability $1 - \epsilon$. Previous work has quantified the benefits of assigning the action-value function with prior knowledge in this way [Hailu and Sommer, 1999].

Ideally, we do not wish to assume such prior knowledge is given but rather have the agent learn it. In the transfer learning setting, an agent would start out with no prior knowledge and would gain this knowledge by solving some source MDPs. Thereafter, the agent would use this attained knowledge and transfer it to a new, related MDP.

As an example of how such transfer can be applied, previous work in transfer learning proposed action-priors [Rosman and Ramamoorthy, 2012]. With action-priors, while solving some source MDPs, the agent keeps track of which actions are optimal in each state. This knowledge can then be transferred to a new MDP by biasing action selection so that instead of taking a random action with probability ϵ, the agent instead selects an action from the action-prior distribution with probability ϵ. Provided the new MDP is similar enough to the source MDPs, this can accelerate learning and improve sample effciency. A suite of other transfer learning methods has been proposed for RL [Taylor and Stone, 2009].

2.5 Aims and Contributions

This thesis takes a step towards addressing the challenges in RL, as described in Section 2.2, by leveraging prior knowledge to improve the sample efficiency of RL algorithms. The main contributions of this thesis are:

- In Chapter 3 we introduce a new reward shaping framework called Belief Reward Shaping (BRS) that can integrate with any model-free RL algorithm, such as Q-learning. The main idea behind BRS is to augment the environment reward distribution with a prior distribution. This prior distribution encodes prior beliefs about a task and can be used to encourage the agent to achieve sub-tasks within a task. Since the agent receives these additional sub task rewards, it is no longer guided only by the sparse environment rewards.

  Therefore, provided the prior distribution encodes knowledge that is useful to solve the task, this leads to improved sample efficiency. Furthermore, we show theoretically that under certain conditions BRS preserves policy invariance so that the set of optimal policies learned with BRS is the same as if we had simply used the sparse environment reward. This property of BRS is important as it means we can specify prior knowledge through the prior distribution to bias the agent without worrying that doing so will have unintended consequences to the behaviour we want the agent to learn.

- In Chapter 4 we introduce a new object-oriented formalism for model-based RL, the Deictic Object-Oriented MDP (Deictic OO-MDP). The Deictic OO-

MDP formalism is based on dietic predicates which are predicates that are grounded with respect to a reference object that relates itself to non-grounded object classes.

Using this formalism, we can show that, for some domains, it is possible to efficiently learn a transferable model of the transition dynamics for the entire domain. This improves sample efficiency because, given a new task from the domain, an agent can transfer the transition model without having to waste samples learning it for the new task.

Furthermore, while previous work on object-oriented representations in RL focus on learning a model of the transition dynamics and assumes the agent knows the reward dynamics, in this thesis we present an algorithm to efficiently learn a transferable model of the reward dynamics as well.

- In Chapter 5 we introduce a new framework to efficiently learn a likely-admissible heuristic function that can then be used with heuristic-based planning algorithms like A*. As discussed in Section 2.2.2 the curse of dimensionality makes value iteration infeasible for practical applications due its high computational complexity. An alternative approach is to use heuristic-based search algorithms. Such algorithms are guided by a heuristic function that encodes prior knowledge about the task. A well-specified heuristic function can greatly reduce computational complexity.

  Many applications in heuristic-based planning assume the heuristic is given. However, it is possible to learn a heuristic function from some source tasks of a domain and then transfer it to solve a new target task of the domain. In this thesis we propose a method to learn such a heuristic in a sample efficient way. Our approach works by generating source tasks in a structured manner by seeking states with high epistemic uncertainty and generating tasks with these start states. Solving these tasks reduces the epistemic uncertainty of the heuristic for these states, thus allowing the next generated batch of source tasks to be different from previously solved source tasks. This approach improves sample efficiency because the algorithm is always generating source tasks that have value in terms of making learning progress.

  Additionally, it is well-known that admissible heuristics lead to optimal plans with many planning algorithms such as A*. In this thesis we introduce a formal framework that utilises epistemic and aleatoric uncertainty to learn a heuristic function that is admissible in a probabilistic sense. Using such a heuristic guides the planner to produce likely-optimal plans with low suboptimality.

Chapter 3

Belief Reward Shaping

The work presented in this chapter also appears in: Marom and Rosman [Marom and Rosman, 2018a].

3.1 Introduction

As discussed in Section 2.2, one of the major challenges in Reinforcement Learning (RL) is sparse rewards. Sparse rewards lead to sample inefficiency because the agent gets little feedback from its environment on how well it is performing at the task while learning to solve it.

A seemingly obvious solution to this problem is to redesign the reward dynamics to be less sparse. Unfortunately, non-sparse rewards in RL can lead to unintended consequences where the agent learns an optimal policy for a fundamentally different MDP that does not produce the intended behaviour of the designer [Clark and Amodei, 2016, Randløv and Alstrøm, 1998].

Interestingly, it has been demonstrated from experiments that if humans are asked to provide numerical feedback to an agent to solve a task, they produce non-sparse reward signals that encourages the agent to achieve sub-tasks [Ho et al., 2015]. This illustrates that, internally, humans may follow non-sparse reward structures to solve tasks, and it is highly plausible that this aids in their efficiency.

One promising way proposed in the literature to overcome the issue of sparse rewards in RL is through the notion of reward shaping. With reward shaping, the agent receives an additional shaping reward that is intended to shape their behaviour in addition to the sparse environment reward. Reward shaping methods are extremely useful in RL because they provide a way of including prior knowledge into an RL algorithm by rewarding the agent for achieving sub-tasks. However, using an ar-

bitrary reward shaping function can lead to the same unintended consequences as with redesigning the environment reward to be less sparse [Randløv and Alstrøm, 1998].

A major breakthrough in the field of reward shaping came from the introduction of Potential-Based Reward Shaping (PBRS) [Ng et al., 1999]. PBRS limits the expressiveness of the reward shaping function that can be given to the agent, but has the benefit of guaranteeing *policy invariance*. Policy invariance is a property whereby the set of optimal policies of a Markov Decision Process (MDP) with and without the inclusion of reward shaping are identical. Policy invariance is an important property of any reward shaping framework because it guarantees that the inclusion to the shaping reward does not lead to unintended behaviours of the agent. After the introduction of PBRS, a variety of other potential-based frameworks were introduced to allow for more expressive shaping rewards to be given to an agent while still maintaining policy invariance [Wiewiora et al., 2003,Harutyunyan et al., 2015,Devlin and Kudenko, 2012].

3.1.1 Contributions

In this chapter we introduce a new framework for reward shaping called Belief Reward Shaping (BRS) that takes a different approach to that of potential-based reward shaping frameworks.

Instead of giving the agent an additional shaping reward, we propose to use Bayesian methods and impose a prior on the environment reward distribution directly. This prior distribution encodes prior knowledge about the task and can bias the agent to achieve sub-tasks. Since, with Bayesian methods, the likelihood overcomes the prior given enough samples, convergence to the environment reward distribution can occur within arbitrary precision. As a result, we can prove formally that under certain conditions BRS preserves policy invariance.

The main advantage of BRS over potential-based frameworks is that, while all potential-based methods restrict the expressiveness of the shaping reward that can be given to the agent, BRS is fully expressive allowing the designer to encode prior knowledge using the complete transition information (s, a, s') directly.

One major advantage of potential-based reward shaping frameworks over BRS is that potential-based frameworks are naturally robust to cyclical behaviour whereby the agent enters into an infinite cycle to receive high sub-task shaping rewards, rather than completing the actual task intended. To overcome this, we propose a modified version of BRS called Delayed Belief Reward Shaping (DBRS) that also has policy invariance but is robust to cycles. With DBRS the agent receives a

shaping reward for a transition only if it has not visited that transition recently, and otherwise receives the sparse environment reward. This mitigates cycles because the agent can no longer exploit the framework by repeatedly getting the shaping rewards through cyclical behaviour.

We run experiments on two tasks, the cliff-jump gridworld and backgammon, to illustrate BRS empirically and show that if the right prior knowledge is encoded we can achieve improved sample efficiency that outperforms competing methods.

3.1.2 Chapter Structure

This chapter is organised as follows: in Section 3.2 we introduce the relevant background on rewards and reward shaping in RL; in Section 3.3 we introduce the BRS framework and provide theoretical guarantees of policy invariance; in Section 3.4 we conduct experiments on the cliff-jump gridworld and backgammon tasks showing the benefits of our proposed framework empirically; and we end the chapter with a discussion of future work in Section 3.5 and concluding remarks in Section 3.6.

3.2 Background

3.2.1 Rewards in Reinforcement Learning

As discussed in Section 2.1.1 an RL task can be described as an MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where $\mathcal{R}(s, a, s')$ is the reward dynamics that govern the reward signal that the agent receives from interacting with its environment.

Since an RL agent aims to learn a policy that maximises its return, the definition of the reward dynamics plays a pivotal role in the form of the optimal policy is learned by the agent. In principle, the reward dynamics should not have to be crafted, but should rather be a natural by-product of the environment. For example, if a human were to touch a hot stove, they would feel a strong sensation of pain. Such a sensation is synonymous with a negative reward in RL. Therefore, if we wanted to construct an RL agent that avoids touching hot objects we would give the agent a negative reward for touching a hot stove. But what should this reward be? Should we set it to -10, -100 or -1000. Unfortunately, there is no natural way to translate from the world of sensations to a numerical value. Therefore, in practice, the reward dynamics have to be carefully crafted by a designer.

In many RL tasks, the only way to craft the reward dynamics to get the intended behaviour for the agent is to make the reward signal sparse. A sparse reward signal is one where the agent gets little feedback on how well they are performing at

a task while learning to solve it. In episodic tasks, this commonly manifests as a reward structure where the agent gets mostly non-positive rewards with few positive rewards.

To see why a sparse reward is often necessary, suppose we adjusted the reward dynamics of the Sokoban MDP described in Section 2.1.2 to be less sparse by also giving the agent a reward of 1 for getting any box to a storage location. Since the aim of the game is to get all boxes to storage locations, this appears to be a useful sub-task reward to incorporate into the design of the reward dynamics. Unfortunately, adjusting the reward dynamics like this may lead to unintended consequences. The agent may learn an optimal policy that moves the same box in and out of the same storage location over and over, each time earning a reward of 1. This cycle gives the agent infinite an return, which is better than getting a finite return for reaching a terminal state. Therefore, the agent learns and optimal policy for the MDP, but this optimal policy will not reflect the behaviour intended for the agent to learn.

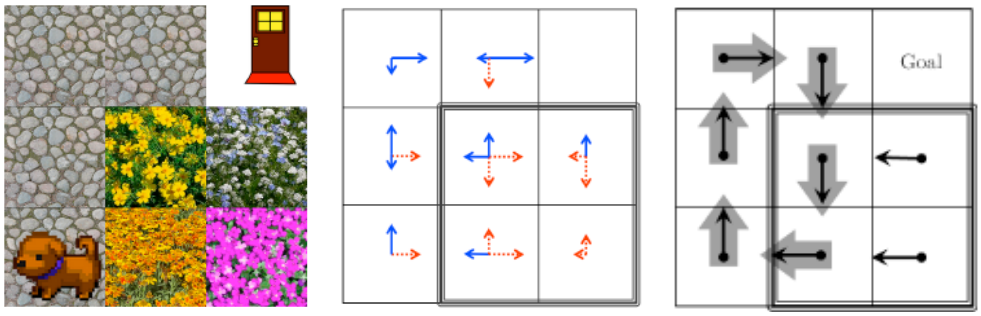
Cycles are not the only problem with sparse rewards. For example, in chess it is often beneficial to take your opponent's queen. In constructing an RL agent to play chess we may therefore determine that it is a beneficial to give the agent a positive reward for achieving this sub-task. However, while in many cases this is a good sub-task to achieve in the game, taking the opponent's queen is not always the optimal move. Therefore, by rewarding this sub-task we force the agent to play in a certain way, rather than letting the agent learn the best way to achieve what is really intended - a win over the opponent.

A good rule of thumb when designing the reward dynamics is that they should reflect *what* we want the agent to accomplish, and *not how* we want the agent to accomplish it [Sutton and Barto, 2018]. If we follow this rule of thumb, a correct way to set up the reward dynamics for an undiscounted Sokoban MDP is to give the agent a positive reward for reaching a terminal state and a negative reward for any other step taken in the environment. This encourages the agent to reach a terminal state in as few steps as possible. While we have illustrated why sparse rewards are necessary for the Sokoban MDP, a similar kind of logic applies to many other RL tasks.

Unfortunately, as discussed in Section 2.2, sparse rewards are a major challenge in RL that lead to sample inefficiency because the agent gets highly delayed feedback for how well they are performing at a task.

3.2.2 How do Humans Assign Rewards?

An interesting observation is that empirical studies suggest that humans have internal reward structures that are non-sparse, and based on accomplishing sub-tasks. This was illustrated in the 'Fido garden-path experiment' [Ho et al., 2015] as shown in Figure 3.1. This task takes place in a gridwold setting where a dog named Fido must learn to get home, but must do so by only walking on paving tiles while avoiding walking on garden tiles.



(a) Fido's garden path task.

(b) Average feedback function.

(c) Cycles induced by human feedback.

Figure 3.1: Fido's garden-path experiment (image source: [Ho et al., 2015]).

This task can easily be solved by an RL agent with sparse rewards. For example, we could design the reward dynamics to give Fido a positive reward of 100 for a step that gets Fido home, a negative reward of -10 for step that leads to a garden tile, and a negative reward of -1 for a step that leads to a paving tile. These reward dynamics encourage Fido to get home in as few steps as possible while avoiding garden tiles.

An experiment was conducted to assess whether humans would naturally construct such reward dynamics for the task [Ho et al., 2015]. In the experiment, 39 human participants were shown a random current location of Fido in the gridworld and told that Fido took some action that resulted in some next location. They were then asked to provide a numerical feedback to give Fido for this transition. The average feedback function provided by the participants is shown in Figure 3.1b where blue arrows represent positive feedback, red arrows represent negative feedback and the magnitude of the arrows represent the intensity of the feedback. We observe that participants tried to help Fido get home by giving him rewards for achieving sub-tasks. For example, large positive rewards are given to Fido for transitioning from one paving tile to another that get Fido closer to home, or if Fido steps from a garden tile and onto a paving tile.

This kind of non-sparse feedback is at odds with the reward dynamics required by an RL agent to solve the task. Furthermore, of the 39 participants in the experiment,

36 of them produced feedback functions that have a net-positive cycle as shown in Figure 3.1c. With these feedback functions it is more rewarding for Fido to circle back along the garden path indefinitely rather than get home, because the sum of rewards across the cycle is net positive.

This experiment illustrates that humans use non-sparse reward structures internally that encourage the achievement of sub-tasks. It is highly plausible that such non-sparse reward structures assist humans to solve tasks efficiently. It therefore makes sense to find methods to incorporate such non-sparse rewards into RL agents to improve their efficiency.

3.2.3 Reward Shaping

One way of overcoming the sparse reward problem in RL is through reward shaping. Reward shaping is a technique whereby the agent receives a shaping reward in addition to the standard environment reward. While the environment reward is viewed as a natural by-product of the environment that cannot be controlled, the shaping reward is viewed as some controllable reward signal given to the agent in order to shape its behaviour. This can be thought of as coming from an external source like a 'trainer' or an internal source like an intrinsic motivation system.

In the most general case, the shaping reward that the agent receives, F, is a function of the complete transition information (s, a, s'), i.e. the agent receives a shaping reward of $F(s, a, s')$. So the total reward that the agent receives for the transition (s, a, s') is $r + F(s, a, s')$ where r is sampled from $\mathcal{R}(s, a, s')$.

While the conceptual idea of reward shaping is useful, allowing the shaping reward to be an arbitrary function of (s, a, s') leads to the same challenges that arise with non-sparse rewards. In particular, the set of optimal policies for an MDP without reward shaping, $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, may not be the same as set of optimal policies for an MDP $\mathcal{M}'$ that is identical to $\mathcal{M}$ except that it includes the shaping reward, $\mathcal{M}' = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} + F, \gamma)$ [Randløv and Alstrøm, 1998].

A major breakthrough in the field of reward shaping came with the introduction of Potential-Based Reward Shaping (PBRS) [Ng et al., 1999]. With PBRS, F is restricted to the form $F(s, a, s') = \gamma\Phi(s') - \Phi(s)$ where Φ is called a potential function. It turns out that using this form of reward shaping is both a necessary and sufficient condition for the optimal policies for $\mathcal{M}$ to be the same as the optimal policies for $\mathcal{M}'$ - a property called *policy invariance*.

The intuitive reason why PBRS preserves policy invariance under F is that the sum of shaping rewards forms a telescoping series that cancel out over the trajectory of

the path taken by the agent i.e.

$$\Phi(s_1) - \Phi(s_0) + \Phi(s_2) - \Phi(s_1) + ... + \Phi(s_T) - \Phi(s_{T-1}) = \Phi(s_T) - \Phi(s_0),$$

where $s_T \in \mathcal{S}_{terminal}$. Then so long as $\Phi(s_T) = \Phi(s_0)$, which by convention are assigned a value of zero, we have that the sum of potentials nets to zero.

To see why potential-based reward shaping is useful, let us return to Fido's garden-path task described in Section 3.2.2. A sparse reward function can solve the intended task of Fido getting to its home without stepping on a garden tile. However, a such sparse reward is not sample efficient. We can use PBRS to include a shaping reward that makes the combined reward non-sparse. To do this, we set $\Phi(s) = -10$ for any state s where Fido is on a garden tile, otherwise $\Phi(s) = 0$. Using this potential function, if Fido is on a garden tile and then steps on a paving tile then Fido will receive a shaping reward of $\Phi(s') - \Phi(s) = 0 - (-10) = 10$. Therefore, the total reward Fido gets for this transition is $R(s, a, s') + F(s, a, s') = -1 + 10 = 9$ and so using PBRS with this potential function makes the combined reward dynamics less sparse. Importantly, using this form of F preserves policy invariance so the behaviour learned by Fido will be as intended.

The major drawback of PBRS is that the potential function Φ can only depend on a state s. This limits the expressive power of PBRS. For example, it may be useful to incorporate information about the action taken by the agent, or the relationship between the current state s and the next state s'. Researchers have addressed this problem with PBRS by proposing other forms of potential-based reward shaping frameworks that are more expressive.

Potential-Based Advice (PBA) [Wiewiora et al., 2003] is a framework that incorporates actions into the potential function i.e. with PBA can we use a potential function $\Phi(s, a)$. However, potential-based advice requires that the agent follows the adjusted policy

$$\pi(s) = argmax_{a \in \mathcal{A}(s)}(Q_*(s, a) + \Phi(s, a))$$

in order to preserve policy invariance. PBA improves upon the expressive power of PBRS but is still limited because the shaping function remains a difference of potential functions. Building on this, later work in reward shaping introduced a framework called Dynamics Potential-Based Advice (DPBA) whereby an arbitrary shaping reward function of the form $F(s, a)$ can be learned in conjunction with learning an optimal policy [Harutyunyan et al., 2015]. Under some technical conditions,

DPBA can also be shown to preserve policy invariance.

Reward shaping is a powerful way of incorporating prior knowledge into model-free RL algorithms such as Q-learning. If this prior knowledge is well-specified, it can be shown to improve sample efficiency by making the total reward that the agent receives less sparse.

3.2.4 Alternatives to Reward Shaping

Reward shaping is not the only way in RL to provide more frequent positive rewards or to encourage the agent to achieve sub-tasks. In this section we highlight alternative techniques at a high-level.

In Hierarchical RL we decompose a task into sub-tasks and learn to solve each sub-task individually [Dietterich, 1998, Sutton et al., 1999]. Specifically, the MAXQ framework uses pseudo-rewards to learn an optimal policy per sub-task, which is analogous to providing more frequent rewards to the agent. Other techniques proposed include biasing the initial values of the action-value function using prior knowledge [Hailu and Sommer, 1999, Matignon et al., 2006], using progress estimators to supply the agent with partial, goal-specific advice [Mataric, 1994, Matignon et al., 2006], or using action priors so that the agent can positively bias action selection in the initial stages of learning [Rosman and Ramamoorthy, 2012]. Intrinsically motivated RL [Singh et al., 2004] is another approach to providing more frequent rewards to the agent. With intrinsically motivated RL the agent receives two distinct types of rewards: an external environment reward that is task-specific, and an intrinsic reward that is task-independent and that comes from an internal motivational system.

An appealing property of reward shaping is that it is easily implementable with model-free based RL algorithms, such as Q-learning. Hierarchical RL requires that a task can be decomposed into sub-tasks. However, for many tasks it is not clear how to achieve such a decomposition. For example, consider the game backgammon where the next set of possible moves depends on a dice roll. This kind of stochasticity makes the game non-trivial to decompose into sub-tasks, as it is impossible to know what the exact game state will be many moves ahead from the current game state.

Biasing the initial action-value function is possible in tabular methods where the action-value function stores a value for each visited state-action pair. However, for tasks with large state-spaces such data structures become infeasible. Instead, we must resort to function approximators, such as linear methods or neural networks, that can generalise from a small subset of observations. Since such function approximators rely on a set of weights, that are usually randomly initialised, it is difficult

to assign them in such a way so as to bias specific states-action pairs.

3.3 Belief Reward Shaping

In this thesis we present a novel framework for reward shaping using Bayesian methods, which we call Belief Reward Shaping (BRS). The advantages of BRS over potential-based frameworks is that:

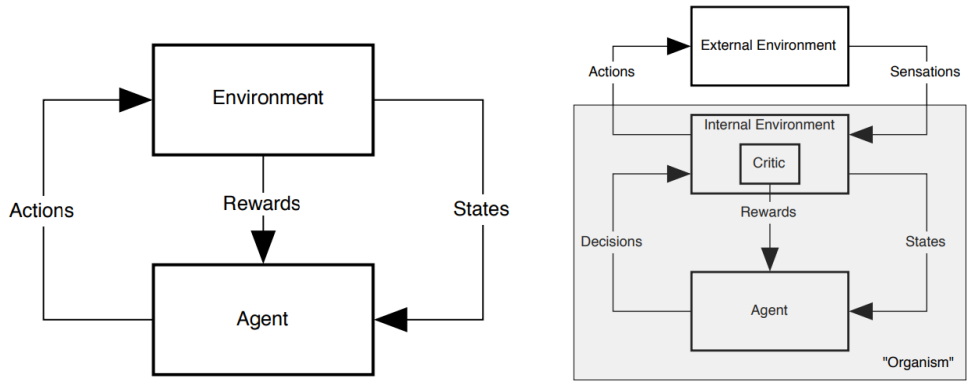
- We can define shaping rewards on the full transition information (s, a, s'). This makes BRS more expressive than even DPBA that can define a shaping rewards on (s, a).
- The shaping rewards can be defined directly. While this is also true for PBRS and PBA, the more expressive DPBA framework requires learning of the shaping reward in conjunction with learning an optimal policy.

Furthermore, under suitable conditions, we can prove that BRS maintains the important policy invariance property that underlies potential-based frameworks.

The departure point between BRS and standard RL is that we argue that rewards should not come only from the environment, but should also incorporate prior beliefs. In the simplified view of RL the agent receives a reward from its environment directly as shown in Figure 3.2a. However, this simplified view fails to capture the internal system of the organism. For example, consider a human that touches a hot stove. The environment produces a sensation which is then processed by the human's nervous system to generate a feeling of pain. This elaborated view is shown in Figure 3.2b, where an internal critic has been introduced within an internal environment that processes sensations that come from the external environment to produce the reward that the agent receives. We emphasise that this elaborated view is not an extension of standard RL, but rather that the simplified view omits these details to reduce presentational complexity [Singh et al., 2004, Sutton and Barto, 2018].

With BRS, we update this elaborated view to account for beliefs. In particular, we argue that the reward that the critic gives the agent should not come only from the environment sensation but also from the critic's belief system. To motivate this view, consider a human that is currently feeling a sensation of pain. This sensation is coming from doing strenuous exercise. However, this negative sensation does not stop them from continuing the exercise because they have a belief that doing this exercise is healthy. Therefore, the human's internal reward does not come solely from the current environment sensation, but also from their beliefs about the activity.

Therefore, in our view, the critic still receives a sensation from the environment.



(a) Simplified view of agent-environment interaction in RL.

(b) Elaborated view of agent-environment interaction in RL (image source: [Singh et al., 2004]).

Figure 3.2: Different views of RL.

Now though, the sensation updates the critic's belief system, and only thereafter the critic provides a reward (which we call the belief reward) to the agent that is an integration of the critic's prior beliefs and the current sensation. This view is illustrated in Figure 3.3.

Whereas in traditional reward shaping frameworks the shaping reward is augmented to the environment reward distribution through the addition of a shaping function, in our framework we augment the shaping reward through a prior distribution on the environment reward distribution directly. These prior beliefs can then be provided to the critic as a form of intuition or advice for a task.

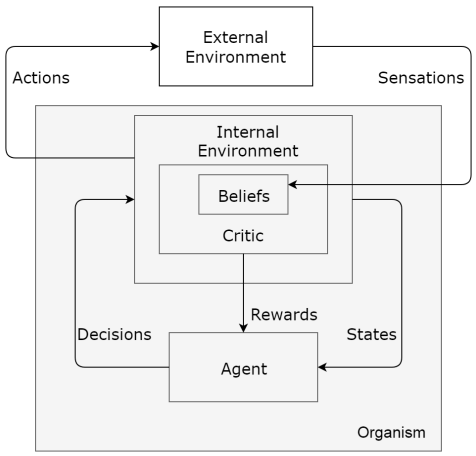


Figure 3.3: The agent / environment interaction with BRS.

3.3.1 Framework

Our departure point from standard RL is that we allow for prior beliefs to be augmented into $\mathcal{R}$. We hereafter refer to the standard RL reward as the 'environment reward' and the reward that the critic gives the agent as the 'belief reward'.

We define a transition as a tuple (s, a, s') where s is a state, a is the action taken when in s and s' is the next state. We denote by $\mathcal{T}$ the set of all possible transitions in the MDP. We denote by $\mathcal{D}_t^\tau$ the history of environment rewards received by the critic for the transition $\tau \in \mathcal{T}$ up to and including time t. For each τ the (unknown) environment reward distribution is given by $p^\tau(r)$. We make the following assumptions in our framework:

- For each τ the critic hypothesises some models of the environment reward distribution $\mathcal{F}^\tau = \{\hat{p}^\tau(r|h) : h \in \mathcal{H}^\tau\}$, where $\mathcal{H}^\tau$ denotes the hypothesis space for transition τ. The $\hat{}$ notation indicates that $\hat{p}^\tau$ is an hypothesis for the true environment reward distribution.
- For each τ and at every $t \geq 0$ the critic knows its prior beliefs represented as a distribution over $\mathcal{H}^\tau$, $q^\tau(h|\mathcal{D}_{t-1}^\tau)$.
- The environment reward received by the critic at time t after τ is identically and independently distributed (i.i.d) from the previous environment rewards received in τ, $\mathcal{D}_{t-1}^\tau$.

The intuition behind our framework follows Bayesian reasoning. Each $\hat{p}^\tau(.|h) \in \mathcal{F}^\tau$ is a possible reward distribution that may be representative of the true environment reward distribution. The critic's belief over possible reward distributions is captured by a prior distribution $q^\tau(h|\mathcal{D}_{t-1}^\tau)$ which depends on the environment rewards the critic has observed so far. Before we start solving the RL task q^τ represents the critic's beliefs without having seen any environment rewards. Therefore, at this point the distribution is influenced only by prior knowledge.

This knowledge may bias certain transitions and hence shape the behaviour of the agent. During learning each time the agent visits transition τ, data is collected from the environment and the critic refines its beliefs over the possible reward distributions and gets closer to the environment reward distribution.

We now formalise this intuition mathematically. At time t, the agent passes through transition τ and the critic receives some environment reward r_t. The critic updates its beliefs using Bayes' rule, *posterior* $\propto$ *likelihood* $\times$ *prior* as

$$q^\tau(h|\mathcal{D}_t^\tau) \propto \hat{p}^\tau(r_t|h) q^\tau(h|\mathcal{D}_{t-1}^\tau). \tag{3.1}$$

The maximum a posteriori (MAP) estimate, $h_{MAP}^\tau = \arg\max_{h \in \mathcal{H}^\tau} q^\tau(h|\mathcal{D}_t^\tau)$ is derived. Lastly, the critic generates an unbiased estimate $\hat{\mu}_t^B$ of $\mathbb{E}_{\hat{p}^\tau(r|h_{MAP}^\tau)}[R(s, a, s')]$, the expected value of $\hat{p}^\tau(r|h_{MAP}^\tau)$, and provides this belief reward to the agent.

In practice, if $\hat{p}^\tau(r|h_{MAP}^\tau)$ is known in closed form we can compute the mean of the

distribution directly, otherwise we may sample observations from the distribution and use the sample mean as an unbiased estimate of the true mean.

By convention we define $q^\tau(h|\mathcal{D}^\tau_{-1}) = q^\tau(h)$, that is the prior distribution with no evidence. We note that because of the data invariance property of Bayes' rule together with our i.i.d assumption, the procedure described only requires us to keep track of $q^\tau(h|\mathcal{D}^\tau_{t-1})$ for all $\tau \in \mathcal{T}$ at time t and not the full histories, $\mathcal{D}^\tau_{t-1}$.

As an illustrative example of BRS, suppose $p^\tau(r)$ follows a $\mathcal{N}(0, \sigma^2)$ distribution where $\sigma^2 = 1$ for some τ. We may define $\mathcal{F}^\tau = \{\hat{p}^\tau(r|\mu)$ follows $\mathcal{N}(\mu, \sigma^2) : \mu \in (-\infty, \infty)\}$. Suppose we use a conjugate Normal prior so that $q^\tau(\mu)$ follows a $\mathcal{N}(\mu_0, \sigma_0^2)$ distribution where $\mu_0 = 5$ and $\sigma_0^2 = 1$. Suppose when the agent transitions to τ for the first time it observes an environment reward of $r_0 = 0.2$ sampled from the true environment reward distribution $\mathcal{N}(0, 1)$. Then the critic updates $q^\tau(\mu|\mathcal{D}^\tau_0)$ with parameters μ_0' and $\sigma_0^{2\prime}$ where:

$$\sigma_0^{2\prime} = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}\right)^{-1}$$

and

$$\mu_0' = \sigma_0^{2\prime}\left(\frac{\mu_0}{\sigma_0^2} + \frac{r_0}{\sigma^2}\right)$$

by using the Bayesian update rule for conjugate Normal distributions [Gelman et al., 1995].

Concretely, for the assigned values to the parameters we have $\mu_0' = 2.6$ and $\sigma_0^{2\prime} = 0.5$. Since $q^\tau(\mu|\mathcal{D}^\tau_0)$ is Normal we have that $h^\tau_{MAP} = \mu_0'$. Therefore, when the agent receives a belief reward at time $t = 0$ it will come from $\hat{p}^\tau(r|\mu_0')$ which is distributed $\mathcal{N}(2.6, 1)$. This will generally be higher than the reward recieved from the true environment reward distribution which is $\mathcal{N}(0, 1)$. Therefore, with this setup, BRS biases the agent to reach transition τ. As more data is collected we have that $\mu_0' \to 0$. and so in the limit we converge to the true environment reward distribution $\mathcal{N}(0, 1)$.

3.3.2 Algorithm

To illustrate how BRS can augment existing RL algorithms, the well-known Q-learning algorithm with BRS is shown in Algorithm 4.

Algorithm 4: *BRS Q-Learning*: Q-learning algorithm augmented with BRS for episodic tasks.

initialise $Q(s,a)$ arbitrarily except for $Q(\mathcal{S}_{terminal},.)=0$
for each $\tau \in \mathcal{T}$ select $\mathcal{F}^\tau = \{\hat{p}^\tau(r|h) : h \in \mathcal{H}^\tau\}$ and $q^\tau(h)$
repeat
  start episode at initial state s
  while $s \notin \mathcal{S}_{terminal}$ **do**
    choose action $a \in \mathcal{A}(s)$ using policy from Q
    observe next state s' and reward r
    update $q^\tau(h|D^\tau)$ using equation 3.1
    compute h^τ_{MAP} from $q^\tau(h|D^\tau)$
    generate an unbiased estimate $\hat{\mu}^B$ of $\mathbb{E}_{\hat{p}^\tau(r|h^\tau_{MAP})}[R(s,a,s')]$
    $Q(s,a) \leftarrow Q(s,a) + \alpha(s,a)(\hat{\mu}^B + \gamma \max_{a' \in A(s')} Q(s',a') - Q(s,a))$
    $s \leftarrow s'$
  end
until *convergence*

3.3.3 Theory

In this section we prove that under suitable conditions Q-learning augmented with BRS has the policy invariance property. We first require some definitions:

Definition 1 (ϵ-adjusted reward dynamics)**.** Let $\mathcal{M}$ be an MDP with transitions $\mathcal{T}$ and reward dynamics $\mathcal{R}$. We define the set of ϵ-adjusted reward dynamics $\mathfrak{R}^\epsilon$ of $\mathcal{M}$ for some $\epsilon > 0$ as follows: for some $(s,a,s') \in \mathcal{T}$ let $\mathcal{R}^y(s,a,s') = \mathcal{R}(s,a,s') + y(s,a,s')$ where $y(s,a,s') \in \mathbb{R}$. Then $\mathfrak{R}^\epsilon$ is a set that contains all reward dynamics of the form $\{\mathcal{R}^y(s,a,s') : (s,a,s') \in \mathcal{T} \text{ and } y(s,a,s') \in (-\epsilon,\epsilon)\}$.

Definition 2 ($\mathfrak{R}^\epsilon$ policy invariance)**.** Let $\mathcal{M} = (\mathcal{S},\mathcal{A},\mathcal{P},\mathcal{R},\gamma)$ be an MDP. Denote by $\Pi^*_{\mathcal{M}}$ the set of optimal policies for $\mathcal{M}$. We say that $\mathfrak{R}^\epsilon$ induces policy invariance with respect to $\mathcal{M}$ if $\Pi^*_{\mathcal{M}} = \{\Pi^*_{\mathcal{M}'} | \mathcal{M}' = (\mathcal{S},\mathcal{A},\mathcal{P},\mathcal{R}',\gamma) \text{ for } \mathcal{R}' \in \mathfrak{R}^\epsilon\}$.

We now prove the theorem:

Theorem 1 (policy invariance under BRS)**.** *Let $\mathcal{M} = (\mathcal{S},\mathcal{A},\mathcal{P},\mathcal{R},\gamma)$ be a finite MDP whose reward dynamics have finite variance. Let $\mathcal{T}$ be the set of all possible transitions (s,a,s') in $\mathcal{M}$. Suppose that for each $\tau \in \mathcal{T}$ we have a set of hypothesised models for the environment reward distribution $\mathcal{F}^\tau = \{\hat{p}^\tau(r|h) : h \in \mathcal{H}^\tau\}$, where $\mathcal{H}^\tau$ denotes the hypothesis space for transition τ and further we have a prior distribution over the hypothesis space $q^\tau(h)$. Suppose further that $p^\tau(r) \in \mathcal{F}^\tau$ where $p^\tau(r)$ is the true environment reward distribution. Then the Q-learning algorithm augmented with BRS described in Algorithm 4 converges to an arbitrary optimal policy from $\Pi^*_{\mathcal{M}}$ provided that:*

1. $\sum_{t=0}^{\infty} \alpha_t(s,a) = \infty$ *and* $\sum_{t=0}^{\infty} \alpha_t^2(s,a) < \infty$ *for all* $(s,a) \in S \times A$.

2. *For each $\tau \in \mathcal{T}$ the environment rewards received by the critic are i.i.d.*

3. *The technical conditions required for the MAP estimate of the posterior distribution to be consistent with the correct hypotheses hold for each $\tau \in \mathcal{T}$. Most notably that the likelihood is a continuous function of the hypothesis space and that the correct hypothesis is not on the boundary of the hypothesis space. For full details of conditions see [Gelman et al., 1995].*

4. *There exists some (arbitrarily small) $\epsilon > 0$ such that $\mathfrak{R}^\epsilon$ induces policy invariance with respect to $\mathcal{M}$.*

Proof. We have assumed that $p^\tau(r) \in \mathcal{F}^\tau$ and that the technical conditions required for the MAP estimate of the posterior distribution to be consistent with the correct hypothesis hold for all $\tau \in \mathcal{T}$. Therefore, by Bayesian asymptotic consistency, $\lim_{t\to\infty} \hat{p}^\tau(r|h^\tau_{t,MAP}) = p^\tau(r)$ for all r in the support of the distribution $p^\tau(r)$. Equivalently for $\epsilon > 0$, there exists some T such that for all $t > T$

$$|\hat{p}^\tau(r|h^\tau_{t,MAP}) - p^\tau(r)| < \epsilon \tag{3.2}$$

for all r in the support of the distribution $p^\tau(r)$. The conditions for Q-learning convergence are assumed to hold in $\mathcal{M}$. Therefore

$$\sum_{t=0}^{\infty} \alpha_t(s,a) = \infty \quad \text{and} \quad \sum_{t=0}^{\infty} \alpha_t^2(s,a) < \infty \tag{3.3}$$

for all $(s,a) \in S \times A$. Equation 3.3 implies that $\alpha_t(s,a)$ is finite for all $(s,a) \in S \times A$ and all $t \geq 0$. We can rewrite equation 3.3 as

$$\begin{aligned} &\sum_{t=0}^{T} \alpha_t(s,a) + \sum_{t=T+1}^{\infty} \alpha_t(s,a) = \infty \quad \text{and} \\ &\sum_{t=0}^{T} \alpha_t^2(s,a) + \sum_{t=T+1}^{\infty} \alpha_t^2(s,a) < \infty \end{aligned} \tag{3.4}$$

for all $(s,a) \in \mathcal{S} \times \mathcal{A}$. The left summations in equation 3.4 are finite because they are finite summations of finite terms. Therefore $\sum_{t=T+1}^{\infty} \alpha_t(s,a) = \infty$ and $\sum_{t=T+1}^{\infty} \alpha_t^2(s,a) < \infty$ for all $(s,a) \in \mathcal{S} \times \mathcal{A}$. So we have that at $T+1$ the conditions required for Q-learning convergence hold.

We have assumed that $\mathfrak{R}^\epsilon$ induces policy invariance with respect to $\mathcal{M}$ for some

$\epsilon > 0$ but we can choose ϵ arbitrarily small in equation 3.2. Therefore we can find some T such that from $t > T$ we will converge to an optimal policy from the set of optimal policies in $\mathcal{M}$ when using $\hat{p}^\tau(r|h^\tau_{t,MAP})$ for the reward distribution.

□

Condition 4 of theorem 1 requires that we have policy invariance with respect to $\mathcal{M}$ when using any reward dynamics in $\mathfrak{R}^\epsilon$ for some arbitrarily small ϵ. This condition is necessary because convergence of $\hat{p}^\tau(r|h^\tau_{t,MAP})$ to $p^\tau(r)$ occurs within arbitrary precision, but never reaches parity. This condition holds in most practical cases. For example, consider the Sokoban MDP described in Section 2.1.2. If we adjust the reward dynamics of all $R(s,a,s')$ by some arbitrarily small $y(s,a,s') \in (-\epsilon, \epsilon)$ and run standard Q-learning, it will still produce an optimal policy for $\mathcal{M}$.

It is straightforward to show that policy invariance still holds when only a subset of transitions are augmented with BRS. Specifically, the only modification required is to define the scope of transitions to be $\mathcal{T}' \subseteq \mathcal{T}$ and apply standard Q-learning for all other transitions. This is useful because it may be cumbersome to define prior beliefs on all transitions and we can focus on specific transitions that are believed to be important for the task.

A natural question to ask is what happens if the assumption that the environment reward distribution is in the critic's hypothesised set of models is false. Unfortunately, in such a case we lose the consistency guarantees of Q-learning with BRS. Nevertheless, we do have a guarantee that the MAP estimate will converge so as to make the hypothesised distribution as close as possible to the true distribution in terms of Kullback-Leibler divergence. For further details on this asymptotic guarantee see [Gelman et al., 1995].

3.3.4 Deterministic Reward Dynamics

In many RL tasks the environment rewards are known to be deterministic. We can adopt this into the BRS framework by using degenerate distributions. Suppose the environment reward for τ is a constant μ. Define $p^\tau(r) = \mathcal{N}(\mu, \sigma^2)$ where $\sigma > 0$. We impose a conjugate normal prior on μ, $q^\tau(\mu) = \mathcal{N}(\mu_0, \frac{\sigma^2}{\lambda})$ where $\lambda > 0$. Applying Bayes' rule to compute $q^\tau(\mu|r_1 = \mu, r_2 = \mu, ..., r_t = \mu)$ and letting $\sigma \to 0$ converges to a degenerate distribution with point mass at

$$\frac{\lambda}{\lambda + t}\mu_0 + \frac{t}{\lambda + t}\mu. \tag{3.5}$$

The parameter λ is the pseudo-count for the prior mean and controls the rate at

which the critic shifts from the prior mean to the true environment reward.

In the deterministic reward setting one may consider an analogous procedure using traditional reward shaping methods, whereby we define some reward shaping function F and multiply it by a decay parameter so as to ensure that the product converges to 0 in the limit. BRS is more general however as it extends naturally to cases where the environment reward is a distribution and it may be advantageous to set prior beliefs on the various parameters of the distribution.

3.3.5 Belief Clusters

In many real-world applications the state-space is very large. This makes it impractical to define a prior for each transition. However, we can work around this by defining a set of transitions that share some common structure. This is a form of state abstraction within the BRS framework. For example, in a Sokoban task we may decide to group together all transitions whereby the agent moves a box to a storage location, rather having to treat each such transition individually. We can achieve this in the BRS framework with the notion of a *belief cluster*.

Formally, we define a belief cluster as any subset, $\mathcal{T}' \subseteq \mathcal{T}$, where the intersection of the hypothesised models induced by $\mathcal{T}'$ is not empty. That is $\mathcal{F}^{\mathcal{T}'} = \cap_{\tau \in \mathcal{T}'} \mathcal{F}^{\tau} \neq \phi$. The critic may now define a prior on all the models in $\mathcal{F}^{\mathcal{T}'}$, $q^{\mathcal{T}'}(h|\mathcal{D}_t^{\mathcal{T}'})$, where the hypothesis space is now $\cap_{\tau \in \mathcal{T}'} \mathcal{H}^{\tau}$ and $\mathcal{D}_t^{\mathcal{T}'} = \cup_{\tau \in \mathcal{T}'} \mathcal{D}_t^{\tau}$ is the history of environment rewards received by the critic for any transition $\tau \in \mathcal{T}'$. The critic may now update its beliefs using Bayes' rule,

$$q^{\mathcal{T}'}(h|\mathcal{D}_t^{\mathcal{T}'}) \propto \hat{p}^{\mathcal{T}'}(r_t|h) q^{\mathcal{T}'}(h|D_{t-1}^{\mathcal{T}'}). \tag{3.6}$$

It is straightfoward to update theorem 1 and show that if for any $\tau_1, \tau_2 \in \mathcal{T}'$ we have $p^{\tau_1}(r) \in \mathcal{F}^{\mathcal{T}'}$ and $p^{\tau_1}(r) = p^{\tau_2}(r)$ and further, each $\tau \in \mathcal{T}$ belongs to exactly one belief cluster then the Q-learning algorithm augmented with belief clusters still has the policy invariance property. In words, these conditions ensure that all transitions in the belief cluster share the same true environment reward distribution and furthermore that the true environment reward distribution exists in the hypothesis space induced by the belief cluster.

If we have that a transition belongs to more than one belief cluster, then we can enhance our framework by allowing the critic to generate multiple belief rewards in each transition, one for each belief cluster that contains the transition, and take the average of these as the final reward given to the agent. Note that theorem 1, where we treat each transition separately, is a special case of belief clusters where

we assign each transition to its own belief cluster.

3.3.6 Function Approximators

The BRS framework proposed in this chapter extends naturally to the case where function approximators are used to represent the action-value function. For example, in Algorithm 4 we would use the same procedure except for line 11, where we would update the parameters of the function using $\hat{\mu}^B$, whereas in standard RL we would use r. An example of using neural networks with BRS for the backgammon task is illustrated in Section 3.4.3

3.3.7 Cycles

BRS has an advantage over potential-based frameworks of being more expressive. However, it does have a disadvantage of not being robust to cycles. Potential-based methods naturally handle cycles because the sum of potential functions form a telescoping series that sum to zero. BRS does not have this property, and so it is possible to induces cycles that have a sum of net positive belief rewards. This cycle will continue until such time as the belief rewards converges to the true environment reward, and this can hinder the convergence rate to an optimal policy. We propose a simple adjustment to make BRS more robust to cycles that we call Delayed Belief Reward Shaping (DBRS).

Suppose that instead of always giving the agent the belief reward, the critic keeps track of the last visit for a transition τ at time t, $N_t(\tau)$ and then gives the agent a reward at time t of

$$\hat{\mu}_t^{DB} = \hat{\mu}_t^B \mathbb{1}(N_t(\tau) \geq K) + r_t \mathbb{1}(N_t(\tau) < K), \tag{3.7}$$

where $\mathbb{1}$ is the indicator function and $K \geq 0$ is a delay parameter that is chosen up-front for the task. The intuition behind this rule is that if the agent has recently been through a transition then the agent receives the environment reward r_t, otherwise, the agent receives the belief reward $\hat{\mu}_t^B$. This reward structure mitigates cyclical behaviour because the agent is does not get the belief reward when re-entering the same transition in a short space of time. Note that if $K = 0$ then DBRS reverts to standard BRS, as the agent always receives the shaping reward $\hat{\mu}_t^B$ at time t. The larger K is set the more strongly DBRS mitigates cyclical behaviour. However, as K increases the agent also receives shaping rewards less frequently.

Note that updating $N_t(\tau)$ at time t only requires knowledge of $N_{t-1}(\tau)$ and the transition τ at time t. Therefore, we can add this as an additional component to an

MDP and still maintain the Markov property. We emphasise that the agent does not know $N_t(\tau)$, and so this does not form part of the state-space representation that the agent receives from its environment. Rather, it is a component that is known to the critic who uses it to determine the reward to give to the agent.

Under the assumptions of theorem 1, $\hat{\mu}_t^B \to r_t$ as $t \to \infty$. Therefore, $\hat{\mu}_t^{DB} \to r_t$ as $t \to \infty$ as well. As a result, policy invariance when using DBRS is maintained under the same conditions of theorem 1. DBRS also naturally extends to the usage of belief clusters by keeping track of the last visit to any $\mathcal{T}' \subseteq \mathcal{T}$ that make up the transitions of the belief cluster.

A natural question that arises is when one should use DBRS over BRS. This question depends on the task, as well as the form of the belief rewards. In tasks where cycles are impossible for the agent to construct there is no need for DBRS. An example of this is a task like backgammon where the stochasticity from dice rolls makes it impossible for the agent to craft any consistent cyclical behaviour. Some tasks have the potential for cycles but careful design of the belief rewards can mitigate them. For example, in chess a belief reward to take the opponent's queen would not induce cycles since once this sub-task is achieved, it cannot be achieved again in a short space of time. However, a sub-task to move your queen to a certain position on the board can easily induce cycles. From a performance perspective, as we demonstrate empirically in Section 3.4.2, it is better to use BRS over DBRS if no cycles are induced by the shaping rewards. Therefore, a trade-off exists between BRS and DBRS.

3.4 Experiments

In this section we run experiments in the following settings: the cliff-jump gridworld with BRS in Sections 3.4.1; the cliff-jump gridworld with DBRS in section 3.4.2; and backgammon with BRS in Section 3.4.3.

The cliff-jump gridworld task is used as a simple task to illustrate important aspects of BRS, compare sample efficiency to completing methods and conduct ablation studies. Backgammon is a more complex task that illustrates that BRS can integrate with function approximators, in this case a neural network, to improve sample efficiency in domains with large state-spaces.

3.4.1 Cliff-Jump Gridworld with BRS

The main purpose the cliff-jump gridworld task is to show that we can obtain improved sample efficiency with BRS over competing methods because we can use the

full transition information directly to provide shaping rewards to the agent. The experiment is conducted on a 10×10 gridworld. The agent's goal is to get from a fixed start coordinate $(1, 1)$ to a fixed goal coordinate $(10, 10)$. In order to achieve this, the agent needs to jump over a cliff edge so it can get to the side where the goal coordinate is located. A jump only succeeds with some probability p_{jump}. This task is inspired from various video games where a player needs to make a difficult jump which they may fail many times, but that is necessary to complete the level.

The state-space is (x, y) where x and y represents the agent's current x- and y-coordinates respectively ($x, y \in \{1, 2, ..., 10\}$). The agent can take the actions *North*, *East*, *South* and *West* which move the agent in those directions in the gridworld. In addition, there is a *Jump* action. If the *Jump* action is taken when the agent is on the edge of a cliff then the agent is transported to the other side with probability p_{jump}, otherwise the agent 'falls off' the cliff and is transported back to the start coordinate. If the jump action is taken in any other state, the agent remains in the current state. Furthermore, if the agent takes a *North* or *South* action that moves it off the cliff then it is also transported back to the start coordinate. Finally, any action that should take the agent off the gridwold leaves the state unchanged. The gridworld is illustrated visually in Figure 3.4 where the start coordinate is denoted by S, the goal coordinate by E, and the cleft that the agent must jump over is denoted by C (i.e. the cliff edges are at $y = 2$ and $y = 4$).

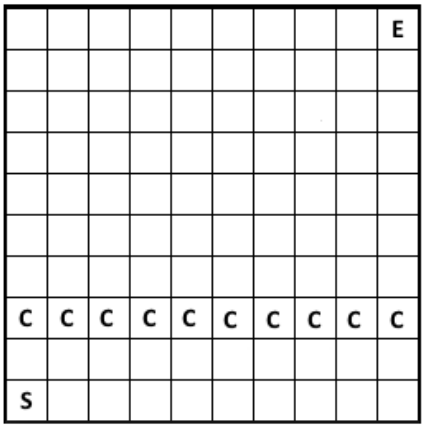


Figure 3.4: The cliff-jump gridworld task.

The environment rewards are -1 for every step that is non-terminal and 100 for a step that leads to a terminal state. For learning to solve this RL task we use an ϵ-greedy policy with $\epsilon = 0.1$, $\gamma = 1$, a constant learning-rate $\alpha = 0.05$ and $p_{jump} = 0.2$. We apply an early termination criterion of 100 steps per episode.

We use equation 3.5 to model the environment rewards as they are deterministic. Let $d(s, c)$ represent the distance from state s to coordinate $c = (x, y)$ for some arbitrary distance metric. Specifically, in our experiments we use the Manhattan

distance. Intuitively, we want our agent to get to the cliff edge as quickly as possible, take the jump and then get to the end coordinate as quickly as possible. Define $s.x$ and $s.y$ to be the x- and y- coordinates in s respectively, $c_1 = (1, 2)$ (the bottom cliff edge coordinate) and $c_2 = (10, 10)$ (the end coordinate). Then to encode this prior knowledge we first define the following belief clusters:

- $B_1 = \{(s, a, s') \in \mathcal{T} : \ s.y \leq 2, \ d(s', c_1) < d(s, c_1)\}$
- $B_2 = \{(s, a, s') \in \mathcal{T} : \ s.y \leq 2, \ d(s', c_1) \geq d(s, c_1)\}$
- $B_3 = \{(s, a, s') \in \mathcal{T} : d(s, c_1) = 0, \ a = \mathit{Jump}\}$
- $B_4 = \{(s, a, s') \in \mathcal{T} : \ s.y > 2, \ d(s', c_2) < d(s, c_2)\}$
- $B_5 = \{(s, a, s') \in \mathcal{T} : \ s.y > 2, \ d(s', c_2) \geq d(s, c_2)\}$

We will set priors on B_1 and B_2 to encourage the agent to get to the edge as quickly as possible, on B_3 to encourage the agent to take a jump at the edge and on B_4 and B_5 to encourage the agent to get to the end coordinate as quickly as possible once it has reached the other side of the cliff. We set a positive prior mean for belief clusters B_1, B_3 and B_4 as we want to encourage these transitions and negative prior mean for B_2 and B_5 as we want to discourage these transitions.

We compare our results to PBRS and PBA frameworks. While early work on these methods used negative potential functions for distance-based shaping rewards, recent work has shown that positive potentials may improve performance [Grzes and Kudenko, 2009]. We conducted experiments with both negative and positive potentials and found positive potentials had better performance in this task, hence we report these results in this chapter. Tables with results for both positive and negative potentials are reported in Appendix Section A.1.

We note that for our comparisons we have not biased any of the frameworks as they have access to the same level of information and exploit that information to the best of their capabilities. In total we run the following algorithms on this task:

- QL: standard Q-learning.
- BRS(μ_0, λ): Q-learning with BRS using belief clusters B_1, B_2, B_3, B_4 and B_5. We set the sign of μ_0 positive for belief clusters B_1, B_3 and B_4 and negative for B_2 and B_5 while the same λ is used for all belief clusters.
- PBRS(μ_0, μ_1): Q-learning with PBRS, $\Phi(s) = \mu_0(\max_{s^* \in S} d(s^*, c_1) - d(s, c_1))\mathbb{1}(s.y \leq 2) + \mu_1(\max_{s^* \in S} d(s^*, c_2) - d(s, c_2))\mathbb{1}(s.y > 2)$.
- PBA(μ_0, μ_1, μ_2): Q-learning with PBA, $\Phi(s, a) = \mu_0(\max_{s^* \in S} d(s^*, c_1) - d(s, c_1))\mathbb{1}(s.y \leq 2) + \mu_1(\max_{s^* \in S} d(s^*, c_2) - d(s, c_2))\mathbb{1}(s.y > 2) + \mu_2\mathbb{1}(d(s, c_1) =$

$0,\ a = Jump$).

For $\mu_0 > 0$ and $\mu_1 > 0$, the potential $\Phi(s)$ for PBRS encourages the agent to move closer to c_1 when at the bottom of the cleft and to c_2 when at the top of the cleft. To illustrate this, suppose the agent is currently in state $s = (1, 1)$, takes action *East* to reach state $s' = (1, 2)$. We have that $\max_{s^* \in S} d(s^*, c_2) = (10-1)+(10-2) = 17$. Then $\Phi(s) = 17 - 1 = 16$ while $\Phi(s') = 17 - 2 = 15$. Therefore, the shaping reward is $\mu_0(15 - 16) = -\mu_0$. However, if the agent takes action *North* from $s = (1, 1)$ to reach state $s' = (1, 2)$ then $\Phi(s') = 17 - 0 = 17$ and the shaping reward is $\mu_0(17 - 16) = \mu_0$. The potential $\Phi(s, a)$ for PBA is identical to $\Phi(s)$ except that it adds a term to encourage the agent to take the *Jump* action when at c_1.

Each algorithm is tested on a grid of parameter values. BRS(μ_0, λ) is run for $\mu_0 \in \{0.5, 1, 5, 10, 100\}$ and $\lambda \in \{100, 500, 1000, 5000, 10000\}$. PBRS$(\mu_0, \mu_1)$ and PBA(μ_0, μ_1, μ_2) are run where each parameter takes values in $\{0.5, 1, 5, 10, 100\}$. We run each algorithm / parameter set over 1000 episodes and average over 50 independent runs. Plots are then averaged over 10 consecutive points with standard deviation error bars included.

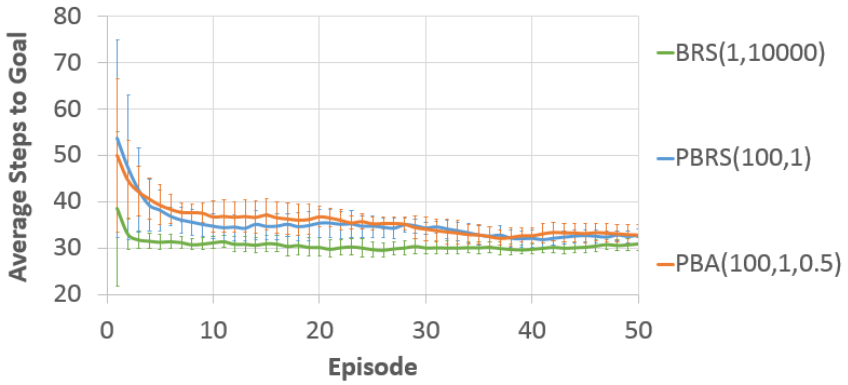


Figure 3.5: Cliff-jump best performance for each algorithm.

We see from Figure 3.5 that using reward shaping is significantly more sample efficient than standard Q-learning for optimal parameter values (Q-learning converges in approximately 700 episodes as shown in Figure 3.7). BRS outperforms other reward shaping methods. In fact, it converges to the optimal policy almost immediately whereas PBRS and PBA converge in approximately 50 episodes. This improvement in sample efficiency when using BRS is unsurprising given that we can be provide shaping rewards using the full transition information directly.

Now consider Figure 3.6. When using BRS with deterministic rewards, we have two parameters - μ_0, which is the prior mean and λ which is the pseudo-count of the prior mean. In practice, λ controls the rate at which the critic shifts our beliefs from the prior mean to the true environment reward, and the lower λ the faster we shift

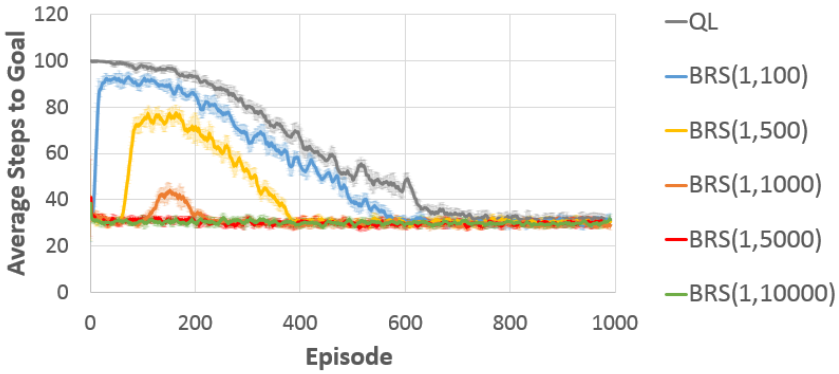


Figure 3.6: Cliff-jump sensitivity to λ.

from the prior mean. In most cases μ_0 is more intuitive to set than λ. In Figure 3.6 we show how BRS performs when we set an intuitive prior $\mu_0 = 1$ for the different values of λ.

We see that even when we set $\lambda = 10^2$, small, we get improvements over standard Q-learning. This is because the agent has the benefit of learning from optimal actions early in the learning process, albeit for a short time. When $\lambda = 10^3$ we have similar convergence to the optimal parameter settings although we have a period of divergence between 150 and 200 episodes. The reason for this is that at the start of learning, the agent's trajectories are controlled most strongly by the priors we set and when this reduces the agent starts exploring based on the environment dynamics. In this task, the agent initially is encouraged by the positive prior μ_0 to jump at the cliff edge but once the environment dynamics start to dominate the agent starts incurring negative rewards for jumping, as the jump fails with high probability, and this leads to exploration of unexplored states. Since these states are not on the trajectory of the optimal policy the agent returns over time to the optimal policy. Overall we see that using BRS with small λ may still provide significant improvements in sample efficiency and that BRS is robust to λ in this task when we set a reasonable value for the prior mean.

In Figure 3.7 we examine results for parameter settings that result in poor sample efficiency. Consider first BRS$(100, 10000)$ where we set $\mu_0 = 100$ high and $\lambda = 10^4$ high. In this case BRS converges to the optimal policy almost immediately but then starts to diverge at around 500 episodes. The reason for this divergence is that the agent enters a cycle whereby it moves in and out of c_1 repeatedly. This is because the prior mean is set so high and the rate at which we shift to the environment reward so slow that the agent learns to prefer obtaining the shaping rewards over solving the underlying RL task. This issue with cycles is similar to that described in Section 3.2.2 with non-sparse reward in Fido's garden path task. However, we

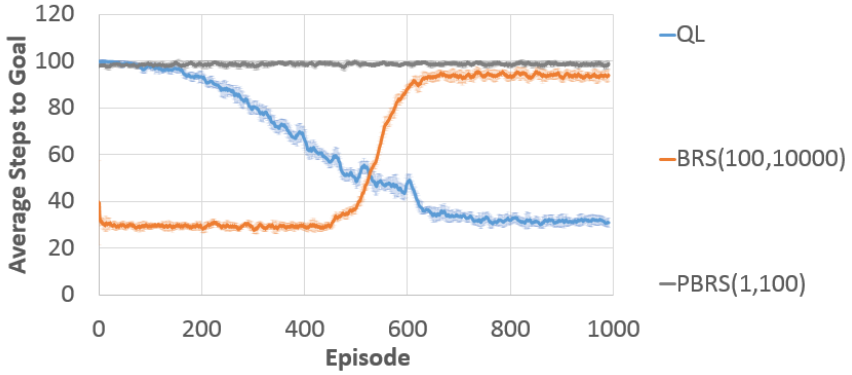


Figure 3.7: Cliff-jump examples of poor performance.

note that with BRS we have convergence in the limit.

Next consider PBRS$(1, 100)$. At 1000 episodes we see the agent is still behaving as poorly as when it started. We conclude from this that poorly specified prior knowledge results in slower convergence to an optimal policy regardless of the method of reward shaping chosen. Furthermore, one may argue that in both the cases of BRS$(100, 10000)$ and PBRS$(1, 100)$ it is not immediately obvious that the chosen parameter values would produce poor sample efficieny. Therefore, care must be taken when choosing shaping rewards in RL tasks to ensure they improve sample efficiency.

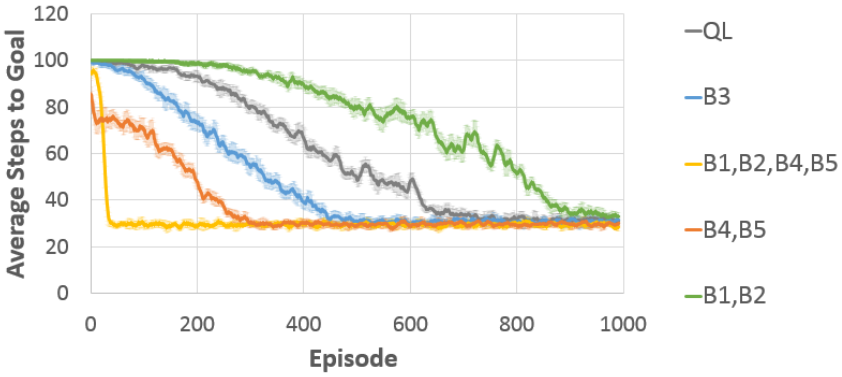


Figure 3.8: Cliff-jump examples with incomplete information.

We run ablation studies to analyse how BRS performs with incomplete information. In Figure 3.8 we show results where we do not use the full set of belief clusters. In all these experiments we use $\mu_0 = 1$ and $\lambda = 10^4$. We see that in most cases we obtain improved sample efficiency, even in the setting of incomplete information. For example, when only using belief cluster B_3 where the prior knowledge incentivises the agent to jump at the cleft we get significant improvements over standard Q-learning. Interestingly, when using only belief clusters B_1 and B_2 we get worse sample efficiency over standard Q-learning. Recall that the priors on these clusters

incentivise the agent to get to the cleft edge as quickly as possible. However, since there is no immediate incentive to jump at the cleft, the agent learns to optimise the shaping reward by walking in and out of coordinate c_1.

More details about the experimental results for this task are available in Appendix Section A.1 where we tabularise the results for each algorithm and parameter value.

3.4.2 Cliff-Jump Gridworld with DBRS

As demonstrated in Section 3.4.1, cycles are induced in the cliff-jump gridworld task if we use only B_1 and B_2 with $\mu_0 = 1$ and λ high. This is because the agent prefers to get the positive shaping reward by cycling in and out of c_1 rather than completing the actual task intended.

We can mitigate this issue of cycles by using DBRS described in Section 3.3.7. To evaluate this we run BRS(μ_0, λ) and DBRS(μ_0, λ, K) on this task with high λ, where BRS(μ_0, λ) is defined as per Section 3.4.1 and DBRS(μ_0, λ, K) is defined in an identical manner to BRS(μ_0,λ) except that we use the DBRS with a delay of K. For our experiments we set $\mu_0 = 1$ and $\lambda = 10^6$. We use $K = 5$ when running DBRS.

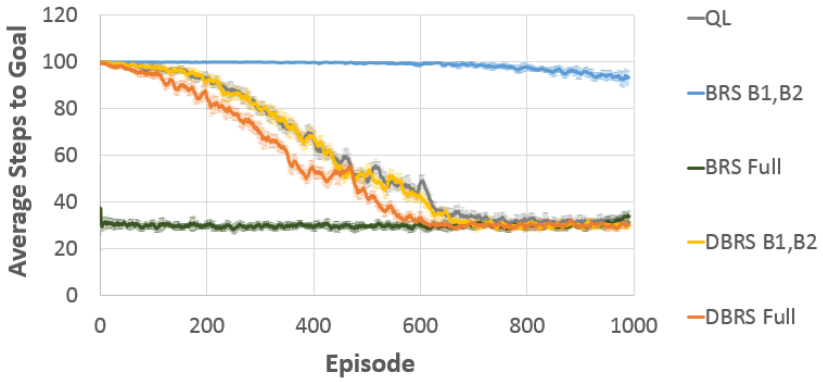


Figure 3.9: Cliff-jump task performance with DBRS.

Results are shown in Figure 3.9. We see from this figure that BRS with only belief clusters B_1 and B_2 performs poorly, and is even outperformed by standard Q-learning because of the cycles induced. When we run DBRS with B_1 and B_2 we get improved performance over BRS that is similar to standard Q-learning performance. When we run DBRS with all belief clusters we get an improvement over standard Q-learning, but are outperformed by BRS run with all belief clusters. This shows that there is a trade-off when using DBRS over BRS. Since BRS does not delay the agent in receiving belief rewards, if the priors are well specified it performs better than DBRS. However, DBRS is more robust to cycles.

We conduct a further experiment to assess the behaviour of DBRS for different values of the delay parameter K. In this experiment use the full set of belief clusters and run DBRS with $K \in \{0, 2, 5, 10, 100\}$.

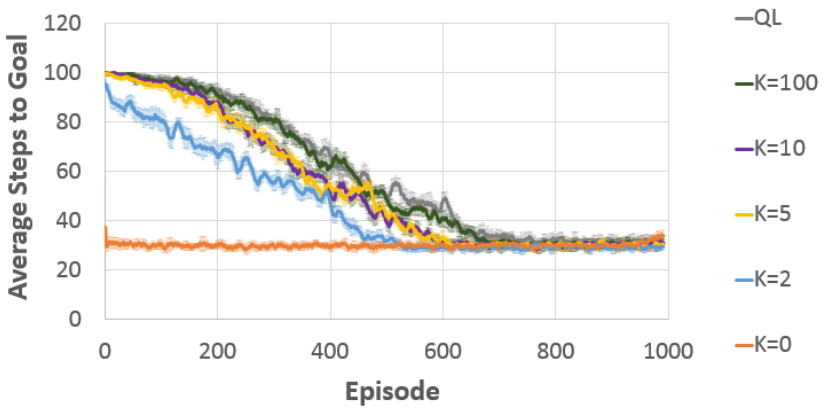


Figure 3.10: Cliff-jump task with different values of K.

Results are shown in Figure 3.10. We see from the results that when $K = 100$ we get performance that is almost identical to standard Q-learning. This is because the agent receives highly delayed belief rewards under a setting where K is large. As K decreases we achieve better performances with DBRS as the agent receives belief rewards more frequently. The case $K = 0$ produces the best performance, and this setting is identical to standard BRS as the agent receives the belief rewards without delay.

3.4.3 Backgammon

We now illustrate the benefits of BRS on a more complex task, backgammon. In this section we do not report results under potential-based frameworks as it would be difficult to draw any objective conclusion on which method performs better due to the task's complexity and hence the number of ways we might construct our shaping rewards.

Backgammon has previously been solved using TD(λ) neural networks [Tesauro, 2004]. The first iteration of the TD-Gammon algorithm (TDG0.0) achieved strong intermediate level of play and further refinements to the algorithm achieved master-level play. In this paper we focus on the first iteration due to its ease of implementation and replication - the solution design is clearly described in the literature [Tesauro, 1995, Sutton and Barto, 2018]. We use an identical setup to TDG0.0 for our experiments. Note that in this task, using techniques that bias the initial value function are difficult to implement because the value function is now parameterised by a set of weights, however, reward shaping is well-suited to such tasks.

Our experimental setup is as follows: we first train a baseline neural network (BL) over $50,000$ games. We then train a new neural network repeatedly over 500 games and let it play 10 batches of 50 games against BL. We do this 100 times so that at the end of the process both neural networks have been trained on the same number of games.

We repeat this procedure for 3 different neural networks: one standard network (SNN), one network that uses a simple set of prior beliefs under BRS (SPNN) and one network that uses a complex set of prior beliefs under BRS (CPNN).

With SPNN we set priors on states that are 'improvements' over the starting game state in the sense that we reward having no blot exposure, more than four points, more than one inner point and a maximum prime of more than one - these rewards are only given when degree of contact is not zero.[1] In our setup, the states for each of these concepts are grouped into a separate belief cluster and we use $|\mu_0| = 0.5$ and $\lambda = 3000$ for all belief clusters - the sign of μ_0 depends on whether we want to incentivise or disincentivise the states in that cluster. With CPNN we scale the reward based on the position (i.e. higher prior mean for having 5 inner points than 2 inner points) and we restrict the belief rewards to situations where the position is beneficial (i.e. having a six-point prime is most beneficial when there is an opponent checker trapped behind the prime). For full details on the setup of the prior beliefs for both SPNN and CPNN see Appendix Section A.2

We note that these strategies are quite crude and, in many cases, not optimal - for example there are situations when it is preferable to take risks and increase blot exposure. However, these states can be thought of as beginner level strategies that will beat a random player but will be outperformed by a skilled player. To illustrate this, we also train two additional neural networks, SCNN and CCNN, that use the same structure as SPNN and CPNN respectively but that provide a *constant environment reward* that is equal to the prior mean.

Figures 3.11 and 3.12 show how each of these networks performs against BL by averaging the number of points won across the batches played at each stage of learning. The standard deviations are also shown in this figure.

We see from the results that SPNN performs better at the initial stages of learning than the standard network, SNN. At $10,000$ games this network is performing as well as SNN is after $17,000$ games and gets to a similar level of play as BL by the end of training. Meanwhile SCNN (simple constant rewards) initially performs better than SPNN but is never able to win more than 35% of the points against BL.

[1] For further details about the rules and terminology for backgammon we refer the reader to P. Magriel and R. M. Roberts [Magriel and Roberts, 2004].

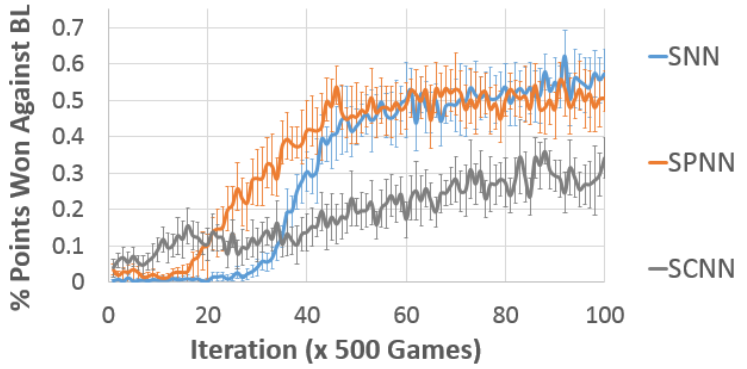


Figure 3.11: Backgammon with simple prior beliefs.

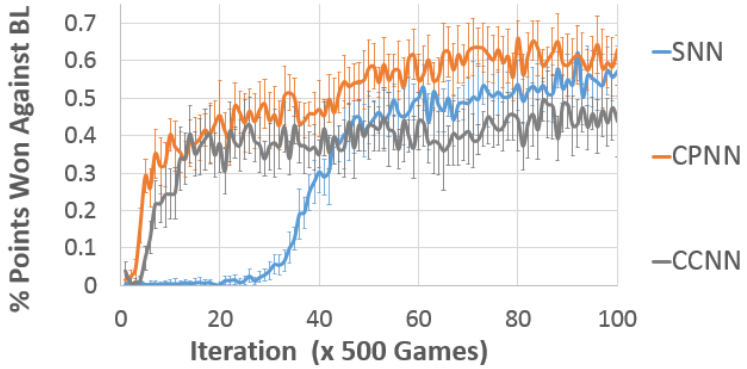


Figure 3.12: Backgammon with complex prior beliefs.

When using CPNN, we get significantly improved performance. At 10,000 games this network wins almost 45% of the points against BL, which took SNN 24,000 training games to achieve and wins 60% of the points by the end of training. CCNN performs better than SCNN did in the previous experiment, but is never able to win 50% of the points or more against BL.

While we do not prove any consistency theorems for TD(λ) neural networks it is reasonable to expect that both SPNN and CPNN will converge to equally strong backgammon agents as SNN because after enough training iterations the belief rewards obtained will be arbitrarily close to the environment rewards. They converge much faster to the because of the shaping rewards provided by BRS. Meanwhile SCNN and CCNN will perform worse than SNN as they indefinitely reward strategies that are not always optimal.

3.5 Future Work

BRS, and reward shaping frameworks in general, incorporate prior knowledge into an RL task in the form of a shaping reward. This shaping reward can be used to

encourage the agent to complete useful sub-tasks within a task. As discussed in Section 1.3, ideally this knowledge should be learned from some previous tasks that the agent has solved and then transferred to new tasks.

In this chapter we have not applied any transfer techniques. Rather, we have assumed all knowledge to be known upfront and shown the benefits of incorporating it into a single task (i.e. cliff-jump gridworld or backgammon). Previous work has shown that in the standard reward shaping setting, where the agent receives a shaping reward $F(s, a, s')$, it is possible to learn a transferable shaping reward function [Konidaris and Barto, 2006, Grzes and Kudenko, 2008]. The fundamental idea to achieve this involves defining an abstract state-space (that is not necessarily Markov) that generalises across tasks of a domain.

As a concrete example of such an abstract state-space, consider the cliff-jump task of Section 3.4.1. When defining the potential function for PBA we encourage the agent to take the *Jump* action at coordinate $c_1 = (1, 2)$. Suppose we want to extend the cliff-jump task to the cliff-jump domain, where the cliff can be placed at any valid location on the grid. In this case, the agent should not necessarily jump at c_1. Therefore, the shaping reward for the cliff-jump task in this chapter is specific to that task, and will not be suitable for other tasks of the domain.

However, we can define an abstract state-space that determines, for any state, the truth values of the propositions *AtCleft* and *OnGoalSide*. The proposition *AtCleft* is true if the agent is at a cleft, and is false otherwise; the proposition *OnGoalSide* is true if the agent is on the goal state side relative to the cliff location, and is false otherwise. Then for any task of the domain, the agent should be encouraged to take the *Jump* action if *AtCleft* is true and *OnGoalSide* is false. Techniques that learn transferable shaping rewards learn a function in such an abstract state-space and transfer it to individual tasks. Applying these ideas to make transfer with BRS viable are outside the scope of this thesis, but are important directions for future research in extending BRS. Furthermore, BRS takes advantage of belief clusters to define a sub-task by grouping together states that share a common structure. While assumed to be known upfront in this thesis, ideally such knowledge should be learned by the agent from previous experience as well.

The notions of using an abstract state-space for transfer as well as learning belief clusters form part of the broader research area of state-abstraction in RL, where the aim is to reduce the size of the state-space of an MDP by grouping together similar states with the aim of reducing learning complexity [Abel et al., 2018].

Finally, BRS and its extension DBRS require multiple hyperparameters to be set. As discussed in Section 3.4 the choice of hyperparameters can have a dramatic

effect on performance, and oftentimes result in poor performance even when the hyperparameters values appear to be reasonably set. Finding techniques for optimal hyperparameter selection in BRS and DBRS is a further important direction for future work.

3.6 Conclusion

This chapter has introduced a new framework of reward shaping called Belief Reward Shaping (BRS). Unlike potential-based reward shaping frameworks that add a shaping reward to the environment reward, BRS operates by including a prior distribution over the environment reward distribution directly. This prior distribution encodes prior knowledge about a task and, if well-specified, can encourage the agent complete useful sub-tasks thus improving sample efficiency.

In terms of the complexity metrics discussed in Section 2.3.1, Q-learning with BRS can be used to reduce learning complexity over standard Q-learning. This is because the agent receives shaping reward when completing useful subtasks, and this can reduce the amount of experience required by the agent to learn an optimal policy.

The main advantage of BRS over potential-based reward shaping frameworks is that BRS is more expressive, allowing us to define shaping rewards on the full transition information directly. BRS also still maintains the policy invariance property that underlies potential-based reward shaping methods.

One disadvantage of BRS over potential-based frameworks is that potential-based methods are robust to cycles. To mitigate this issue under the BRS framework we introduce Delayed Belief Reward Shaping (DBRS) where the agent only receives the belief reward for a transition if it has not recently visited that transition. With DBRS we still have policy invariance under the same conditions of BRS, while DBRS is robust to cycles.

To illustrate the advantages of our framework we ran experiments on a simple cliff-jump gridworld task as well as a more complex backgammon task. In these experiments we showed that with well-specified priors, BRS can improve sample efficiency over competing methods. We also ran experiments with DBRS on the cliff-jump gridworld task showing that, while with BRS it is possible to induce cycles if the priors are poorly specified, DBRS is robust to cycles.

Chapter 4

Zero-Shot Transfer with Object-Oriented Representations

The work presented in this chapter also appears in: Marom and Rosman [Marom and Rosman, 2018b].

4.1 Introduction

As discussed in Section 2.2, one of the major challenges in Reinforcement Learning (RL) is non-transferable representations. This occurs when the knowledge learned by an agent when solving some source task does not transfer to a related target task. This leads to sample inefficient RL because the agent must relearn everything from scratch for each new task it encounters, rather than being able to accelerate learning through prior knowledge gained when solving the source task.

The choice of representation used in RL can lead to significant improvements in transferability. In particular, relational representations [Džeroski et al., 2001, van Otterlo, 2005] are a broad class of representations in RL that have been shown to be effective for the purpose of transfer. Such representations rely on expressing the components of a Markov Decision Process (MDP) in terms of logical statements over variables. Transfer is then achieved by expressing lifted rules about groups of variables, and then grounding these lifted rules for specific instantiations of variable values.

One form of relational representation that has shown much success in RL with regards to transfer is based on viewing the state-space of an MDP as a set of objects, where each object belongs to some object class (or simply class) [Guestrin et al., 2003, Guestrin, 2003, Kansky et al., 2017, Diuk et al., 2008, Diuk, 2010, Scholz et al.,

2014]. Since each object belongs to some class under this representation, it is possible to learn some useful knowledge about an object (or an interaction between objects) and then generalise this to other objects of the same class.

Object-oriented representations are attractive because they bear much resemblance to how humans view the world. For example, in Sokoban a human player would learn how the warehouse keeper and a specific box interact with each other, and then transfer this to each new box encountered in the game. This transferability is possible because the human player infers that boxes are indistinct in the game.

A further appeal of object-oriented representations in RL is that provably efficient learning is possible under this representation [Diuk et al., 2008, Diuk, 2010]. This separates object-oriented representations from other forms of relational representations in RL which, while often more expressive, do not have any provably efficient learning algorithms.

In particular, the Propositional Object-Oriented MDP (Propositional OO-MDP) framework [Diuk et al., 2008, Diuk, 2010] introduces a formalism under which, for certain domains, an optimal policy can be learned under the KWIK (knows what it knows) framework [Li et al., 2008, Li, 2009]. As further discussed in Section 4.2, the KWIK framework introduces a general algorithm, KWIK-R_{max}, to learn a policy for an MDP under the PAC (probably approximately correct) [Kakade, 2003] terms - i.e. the policy produced by KWIK-R_{max} is near-optimal with high probability. The PAC guarantees under KWIK-R_{max} exist provided that the dynamics of the MDP can be learned with polynomial sample complexity under the KWIK protocol.

Domains whose transition dynamics can be represented under the Propositional OO-MDP formalism are ones that can be described as conjunctions of propositional statements over object classes that map to effects over object class attributes. Since the transition dynamics under this representation do not refer to any grounded object, they generalise across an entire domain. Therefore, they can be learned from some source tasks of the domain and then transferred to any new task of the domain.

Propositional OO-MDPs have the advantage of provably efficient learning. However, the restriction that the transition dynamics must be representable only in terms of propositions over object classes limits the framework's applicability. Specifically, the framework does not lend itself to domains where the transition dynamics require to distinguish between different objects of the same class. This can be overcome by using a more expressive form of relational representation, such as the First-Order MDP (FO MDP) [Boutilier et al., 2001] that allows for predicates over grounded objects. However, this complicates both learning and transfer procedures as further

discussed in Section 4.3.4.

4.1.1 Contributions

In this chapter we present a new object-oriented formalism for RL, the Deictic Object-Oriented MDP (Deictic OO-MDP). Deictic OO-MDPs are more expressive than Propositional OO-MDPs, thus allowing for a broader range of domains to be represented under this formalism. While being less expressive than FO MDPs, Deictic OO-MDPs still have the probably efficient learning guarantees that underlies Propositional OO-MDPs and which are not held by FO MDPs.

The core idea behind the Deictic OO-MDP formalism is the notion of a deictic predicate. A deictic predicate is grounded only with respect to a single grounded reference object that must relate itself to non-grounded object classes. Therefore, deictic predicates are more expressive than propositions that may only depend on object classes, while being less expressive that first-order predicates that may depend on an arbitrary number of grounded objects.

This chapter formally outlines the Deictic OO-MDP formalism and introduces a KWIK-learning algorithm to efficiently learn the transition dynamics of any domain that is representable under this formalism. We use two domains to illustrate the Deictic OO-MDP framework: an extension of the classic Taxi domain called the All-Passenger Any-Destination Taxi domain as well as the Sokoban domain. While both these domains are not representable under the Propositional OO-MDP formalism, they are representable under the Deictic OO-MDP formalism.

For both these domains, we show that under the Deictic OO-MDP formalism it is possible to efficiently learn a model of the transition dynamics on a small number of source tasks from the domain and then transfer them to a new task of the domain. In fact, we show that zero-shot transfer is possible so that the agent does not need to do any additional learning once the model has been fully learned on the source tasks. This leads to sample efficiency because the agent can reuse the model for each new task encountered from the domain rather than having to relearn a model from scratch for each new task.

In addition, the original research that introduced Propositional OO-MDPs assumed that the reward dynamics are known and only a model of the transition dynamics needs to be learned by the agent. In the most general case of model-based RL, it is assumed that the agent needs to learn a model of the reward dynamics as well. In this chapter we introduce a propositional object-oriented formalism to efficiently learn the reward dynamics under the KWIK framework. Similar to learning a model of the transition dynamics, this model of the reward dynamics can be learned from

some tasks of the domain and then zero-shot transferred to a new task of the domain.

In totality this chapter shows that, for certain domains, it is possible to efficiently KWIK-learn a model of the transition dynamics (using a deictic representation) and a model of the reward dynamics (using a propositional representation) for the entire domain. These models can be transferred to a new task of the same domain. This improves sample efficiency because the agent can reuse the learned models and does not need to waste samples learning them for each new task.

4.1.2 Chapter Structure

This chapter is organised as follows: in Sections 4.2 and 4.3 we discuss the relevant background for KWIK-R_{max} and Propositional OO-MDPs respectively. In Section 4.4 we introduce the Deictic OO-MDP formalism as well as a KWIK-learning algorithm to learn the transition dynamics under this formalism. In Section 4.5 we introduce a propositional object-oriented formalism to represent the reward dynamics of a domain as well as a KWIK-learning algorithm to learn a of model of the reward dynamics under this formalism. In Section 4.6 we combine the KWIK-learners for the transition and reward dynamics and present a complete KWIK-R_{max} based algorithm called $DOORMAX_D$. In Section 4.7 we conduct experiments on the All-Passenger Any-Destination Taxi domain as well as the Sokoban domain and show that we can learn these models on some source tasks of the domain and then zero-shot transfer them for improved sample efficiency. We end the chapter with a discussion of future work in Section 4.8 and concluding remarks in Section 4.9.

4.2 Background for KWIK-R_{max}

Propositional OO-MDPs introduce a propositional object-oriented formalism that can be used in conjunction with the KWIK framework to efficiently learn the transition dynamics of a domain. In this section we describe the KWIK framework and how it can be used for efficient learning with the KWIK-R_{max} algorithm.

4.2.1 Sample Complexity of Exploration

As discussed in Section 2.3, learning complexity is a measure of how much experience an agent needs to gain in its environment in order to learn an optimal policy. As it turns out, it is often easier to obtain formal guarantees of learning complexity by relaxing the requirement for optimality. One way to measure learning complexity in such a setting is through the definition of *sample complexity of exploration* (or

sample complexity for short) [Kakade, 2003].

Definition 3 (sample complexity)**.** Given a reinforcement learning algorithm **A**, denote by π_t the non-stationary policy executed by **A** at timestep t. For any given $\epsilon > 0$ the sample complexity of **A** is the number of timesteps $t = t'$ such that $\pi_{t'}$ is not ϵ-optimal. That is $V_{\pi_{t'}}(s_{t'}) < V_*(s_{t'}) - \epsilon$ for the state $s_{t'}$.

4.2.2 Probably Approximately Correct MDP

We now define what is meant by a sample efficient RL algorithm under the measure of sample complexity. Intuitively, we prefer algorithms with low sample complexity, which motivates the definition of *Probably Approximately Correct* MDP (PAC-MDP) algorithms [Strehl et al., 2006]:

Definition 4 (PAC-MDP)**.** An algorithm **A** is PAC-MDP in an MDP $\mathcal{M}$ if, given $\epsilon > 0$ and $\delta \in (0, 1)$, the sample complexity of **A** is bounded by a polynomial function in the quantities $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, $\frac{1}{1-\gamma}$ and $|\mathcal{M}|$ with probability $1 - \delta$. The notation $|\mathcal{M}|$ defines some measure of the complexity of $\mathcal{M}$ which is typically a function of the number of parameters describing $\mathcal{M}$ - i.e. the number of states $\mathcal{S}$ and actions $\mathcal{A}$ for finite MDPs.

4.2.3 The R_{max} Algorithm

As it turns out, a number of RL algorithms can be shown to have PAC guarantees [Kearns and Singh, 1998,Brafman and Tennenholtz, 2002,Kakade, 2003,Kearns and Koller, 1999]. One of the most well-known of these algorithms is R_{max} [Brafman and Tennenholtz, 2002], outlined in Algorithm 5. The R_{max} algorithm is a model-based RL algorithm that works on the *optimism in the face of uncertainty principle* described in Section 2.3.3.

With R_{max} the agent initially assumes an (optimistic) model of the environment dynamics such that any action taken in any state leads to a terminal state that produces the highest possible reward r_{max}. The value r_{max} is strictly greater than any return possible for the agent to obtain in the MDP. The agent then applies an exact planning algorithm, like value iteration, to learn an optimal policy under this (incorrect) model. Under this policy the agent is guided to seek these maximally rewarding transitions. In doing so, the agent gains knowledge about the true environment dynamics, updates its model accordingly and then plans again to learn a new optimal policy under the updated model. As the agent gains more knowledge about the true environment dynamics, the optimal policy learned becomes more and more accurate relative to the true environment dynamics.

Algorithm 5: R_{max}: the R_{max} algorithm for episodic tasks.

Input: $\mathcal{S}$, $\mathcal{A}$, γ, $M_{\mathcal{P}}$, $M_{\mathcal{R}}$, r_{max}

initialise $\mathcal{P}_{model}(s'|s,a) \leftarrow \mathbb{1}(s = s')$
initialise $\mathcal{R}_{model}(s,a,s') \leftarrow r_{max}$
initialise $C(s,a) \leftarrow 0$, $C(s,a,s') \leftarrow 0$, $S(s,a,s') \leftarrow 0$
repeat
  start episode at initial state s
  while $s \notin \mathcal{S}_{terminal}$ **do**
    define an MDP $\hat{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}_{model}, \mathcal{R}_{model}, \gamma)$
    compute an optimal policy $\hat{\pi}_*$ for $\hat{M}$ using exact planning algorithm
    choose action $a \in \mathcal{A}(s)$ from $\hat{\pi}_*$
    observe next state s' and reward r
    $C(s,a) \leftarrow C(s,a) + 1$
    $C(s,a,s') \leftarrow C(s,a,s') + 1$
    $S(s,a,s') \leftarrow S(s,a,s') + r$
    if $C(s,a) \geq M_{\mathcal{P}}$ **then**
      foreach $s' \in \mathcal{S}$ **do**
        $\mathcal{P}_{model}(s'|s,a) \leftarrow \frac{C(s,a,s')}{C(s,a)}$
      end
    end
    if $C(s,a,s') \geq M_{\mathcal{R}}$ **then**
      $\mathcal{R}_{model}(s,a,s') \leftarrow \frac{S(s,a,s')}{C(s,a,s')}$
    end
    $s \leftarrow s'$
  end
until *forever*

In the R_{max} algorithm, the parameters $M_{\mathcal{P}} > 0$ and $M_{\mathcal{R}} > 0$ control how many observations we need before we start using the empirical models. The key to making R_{max} PAC is by judicious choice of $M_{\mathcal{P}}$ and $M_{\mathcal{R}}$. In particular, for the finite MDP setting where $\mathcal{P}(s'|s,a)$ is a multinomial distribution the lowest value we can set for $M_{\mathcal{P}}$ to make R_{max} PAC is $\mathcal{O}(\frac{|S|}{\epsilon^2}\ln\frac{|S|^2|A|}{\delta})$ [Li, 2009]. Moreover, in the case of deterministic environments R_{max} actually produces an optimal policy with probability one for $M_{\mathcal{P}} = 1$ and $M_{\mathcal{R}} = 1$. This is because we only need one observation for each input to completely learn the dynamics for that input.

4.2.4 KWIK-Learnable Classes

The R_{max} algorithm outlined in Section 4.2.3 applies to finite MDPs where the representation used for the models are the tuples (s, a, s'). Note that even in the simplest setting of deterministic environments with $M_{\mathcal{P}} = 1$, we need a total of $|S||A|$ unique samples to completely learn the transition dynamics. Due to the curse of dimensionality, even with the benefits of efficient exploration provided by R_{max}, convergence to a near-optimal policy becomes impractical for real-world tasks with large state-spaces.

Using the tuple (s, a, s') to learn the models of the environment dynamics is therefore impractical. A host of other, more compact, representations have been suggested as alternatives. These include: function approximators [Strehl and Littman, 2008], dynamic Bayesian networks [Strehl, 2007, Strehl et al., 2007] and relations [Džeroski et al., 2001, Diuk et al., 2008, Diuk, 2010].

The KWIK framework unifies all these representations through the notion of KWIK-learnable classes [Li et al., 2008, Li, 2009]. Under this framework, if a class of MDPs is KWIK-learnable under the KWIK protocol then it is possible to construct an R_{max} type algorithm for that class of MDP with PAC guarantees. The KWIK protocol is described as follows:

There is an input set $\mathcal{X}$ and an output set $\mathcal{Y}$. A hypothesis class $\mathcal{H}$ is a set of functions mapping inputs to outputs i.e. $\mathcal{H} \subset (X \to Y)$. Then interaction under the protocol proceeds as follows

- The hypothesis class $\mathcal{H}$, an accuracy parameter $\epsilon > 0$, and a confidence parameter $\delta \in (0, 1)$ are known to both the agent and the environment.
- The environment selects a target function $h^* \in \mathcal{H}$ adversarially.
- For timesteps $t = 1, 2, 3, ...$:
  - The environment selects an input $x \in \mathcal{X}$ arbitrarily and informs the agent.

- The target value $y = h^*(x)$ is unknown to the agent.
- The agent either 1) makes a prediction $\hat{y}$ for y or; 2) responds with $\perp$, where $\perp$ indicates that the agent is yet unable to make an accurate prediction of y.
- If $\hat{y} \neq \perp$ then is must be ϵ-accurate i.e. $|\hat{y} - y| < \epsilon$.
- If $\hat{y} = \perp$ then the agent receives a stochastic observation $z \in Z$ of the output y (for example, if y is Bernoulli then $z = 1$ with probability y and $z = 0$ with probability $1 - y$).

We say that $\mathcal{H}$ is a KWIK-learnable class if there exists some algorithm $\mathbf{A}$ such that in a run of the KWIK protocol with probability at least $1 - \delta$:

- If $\hat{y} \neq \perp$ then it must be ϵ-accurate. (*accuracy requirement*)
- The total number of times that the learner may respond with $\perp$ is bounded by a function that is polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$ and some measure of the size of the hypothesis class $\mathcal{H}$. This number is called the KWIK-bound. (*sample complexity requirement*)

4.2.5 The KWIK-R_{max} Algorithm

The KWIK-R_{max} Algorithm [Li et al., 2008, Li, 2009], as outlined in Algorithm 6, generalises the R_{max} algorithm to a broader class of MDPs. The algorithm requires as input two sub-algorithms, $\mathbf{A}_{\mathcal{T}}$ and $\mathbf{A}_{\mathcal{R}}$, that KWIK-learn the transition dynamics and reward dynamics respectively. Given these inputs the KWIK-R_{max} algorithm is guaranteed to be PAC with respect to the precision and confidence parameters for the transition and reward dynamics respectively, $\epsilon_{\mathcal{T}} > 0$, $\delta_{\mathcal{T}} \in (0, 1)$ and $\epsilon_{\mathcal{R}} > 0$, $\delta_{\mathcal{R}} \in (0, 1)$. However, by judicious choice of these parameters, KWIK-R_{max} can be shown to have the PAC guarantees for $\epsilon > 0$ and $\delta \in (0, 1)$ [Li, 2009].

A major convenience of KWIK-R_{max} is that it shifts focus away from the learning algorithm used to learn a policy and puts it on KWIK-learnability. Practitioners may focus on developing algorithms that KWIK-learn the environment dynamics knowing that the KWIK-R_{max} algorithm guarantees a PAC policy regardless of any other factors, such as the choice of representation used.

4.3 Background for Propositional OO-MDPs

The KWIK framework can be used in conjunction with object-oriented representations for efficient learning. In particular, previous work has introduced the Propo-

Algorithm 6: KWIK-R_{max}: the KWIK-R_{max} algorithm for episodic tasks.

Input: $\mathcal{S}$, $\mathcal{A}$, γ, $\mathbf{A}_\mathcal{T}$, $\mathbf{A}_\mathcal{R}$, $\epsilon_\mathcal{T}$, $\delta_\mathcal{T}$, $\epsilon_\mathcal{R}$, $\delta_\mathcal{R}$, r_{max}

run $\mathbf{A}_\mathcal{T}$ with $\epsilon_\mathcal{T}$ and $\delta_\mathcal{T}$
run $\mathbf{A}_\mathcal{R}$ with $\epsilon_\mathcal{R}$ and $\delta_\mathcal{R}$
repeat
  start episode at initial state s
  while $s \notin \mathcal{S}_{terminal}$ **do**
    foreach $(x, u) \in \mathcal{S} \times \mathcal{A}$ **do**
      if $\boldsymbol{A}_\mathcal{T}(x, u) = \perp$ **then**
        $\mathcal{P}_{model}(x'|x, u) \leftarrow \mathbb{1}(x = x')$
      else
        $\mathcal{P}_{model}(x'|x, u) \leftarrow \mathbf{A}_\mathcal{T}(x, u)$
      end
    end
    foreach $(x, u, x') \in \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ **do**
      if $\boldsymbol{A}_\mathcal{R}(x, u, x') \leftarrow \perp$ **then**
        $\mathcal{R}_{model}(x, u, x') \leftarrow r_{max}$
      else
        $\mathcal{R}_{model}(x, u, x') \leftarrow \mathbf{A}_\mathcal{R}(x, u, x')$
      end
    end
    define an MDP $\hat{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}_{model}, \mathcal{R}_{model}, \gamma)$
    compute an optimal policy $\hat{\pi}_*$ for $\hat{M}$ using exact planning algorithm
    choose action $a \in \mathcal{A}(s)$ from $\hat{\pi}_*$
    observe next state s' and reward r
    if $\boldsymbol{A}_\mathcal{T}(s, a) = \perp$ **then**
      update $\mathbf{A}_\mathcal{T}$ with sample (s, a, s')
    end
    if $\boldsymbol{A}_\mathcal{R}(s, a, s') = \perp$ **then**
      update $\mathbf{A}_\mathcal{R}$ with sample (s, a, s', r)
    end
    $s \leftarrow s'$
  end
until *forever*

sitional Object-Oriented MDP (Propositional OO-MDP) formalism and showed if a domain can be represented under this formalism, it is possible to KWIK-learn the transition dynamics of that domain. In this section we discuss the related background for Propositional OO-MDPs.

4.3.1 Relational Reinforcement Learning

Relational RL encompasses a broad class of representations for RL that are based on viewing components of an MDP in terms of the relational structure between variables. Such a relational structure is typically expressed through logical statements (such as propositions or first-order predicates) over these variables. In relational RL, lifted relations express the relational structure between groups of variables, while grounded relations express the relational structure between specific instantiated variable values.

Relational representations incorporate prior knowledge into RL tasks because, generally, some information about the type or structure of the relations is assumed to be known upfront. However, the benefit of this approach is that it produces a more compact representation of an MDP that is more amenable to transfer. This is because the lifted relations generalise across instantiations of grounded relations.

As an illustrative example of the benefits of Relational RL, we can consider the Sokoban domain. An example of how to encode a Sokoban task as an MDP in a non-relational way was discussed in Section 2.1.2, while in Section 2.2 we discussed the challenges with such a representation in terms of transfer.

Suppose we have a predicate $Touch_E(p_x, b_x)$ that takes in two variables: p_x, the x-coordinate of a warehouse keeper; and b_x, the x-coordinate of a box. The predicate $Touch_E(p_x, b_x)$ then returns a truth value of 1 if $b_x = p_x + 1$ - that is, the box is one square east of the warehouse keeper in the x-coordinate space - and 0 otherwise.[1] Suppose we have a similar predicate $Touch_E(b_x, w_x)$ that is defined the same as $Touch_E(p_x, b_x)$, except that it takes in as arguments x-coordinate of a box and wall respectively. These are lifted logical statements because they have not been instantiated with any specific variable values.

Then the transition dynamics of any Sokoban task can be partially described by taking conjunctions over these predicates. In particular, for grounded instantiations of the variables $p_x = x_1$, $b_x = x_2$ and $w_x = x_3$ we have that if

$$Touch_E(x_1, x_2) = 1 \wedge Touch_E(x_2, x_3) = 1$$

[1]For clarity, a truth value of 1 implies that the statement is true, while a truth value of 0 implies that the statement is false.

then the effect of taking the action *East* is

$$x_1 \leftarrow x_1 + 0 \text{ and } x_2 \leftarrow x_2 + 0 \text{ and } x_3 \leftarrow x_3 + 0.$$

In words, if a warehouse keeper has a box one square east of it and that same box has a wall once square east of it, then the warehouse keeper, box and wall shift by zero in the x-coordinate space when the action *East* is taken.

Therefore, if we could learn the lifted rule

$$\mathit{Touch}_E(p_x, b_x) = 1 \wedge \mathit{Touch}_E(b_x, w_x) = 1 \implies p_x \leftarrow p_x + 0, b_x \leftarrow b_x + 0, w_x \leftarrow w_x + 0$$

for action *East*, then this rule would transfer to any Sokoban task allowing us to reuse it in useful ways to accelerate solving new Sokoban tasks.

While the expressive power of relational RL is appealing, efficient learning under relational RL frameworks is challenging. Multiple formalisms have been proposed that limit the expressiveness of the relations in an attempt to make learning more tractable [Džeroski et al., 2001, van Otterlo, 2005, Diuk et al., 2008, Diuk, 2010].

4.3.2 Object-Oriented Representations

One form of Relational RL representation that has shown promise with regards to efficient learning is the object-oriented representation. Object-oriented representations in RL were first introduce under the Relational MDP (RMDP) formalism [Guestrin et al., 2003, Guestrin, 2003] through the notion of a schema (or template). A schema is a lifted representation describing the different components of an MDP - $\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$ and $\mathcal{R}$ - for an entire domain, rather than for an individual task. Thereafter, grounded MDPs representing tasks from the domain can be instantiated from the schema.

Under an object-oriented approach, the state-space for such a schema consists of a set of object classes $\mathfrak{C} = \{C_i\}_{i=1}^{N_\mathfrak{C}}$. Each object class $C \in \mathfrak{C}$ has a set of attributes $\mathit{Att}(C) = \{C.\alpha_i\}_{i=1}^{N_C}$ and each attribute $C.\alpha \in \mathit{Att}(C)$ of an object class has a domain of possible values $\mathit{Dom}(C.\alpha)$. Given a schema, a grounded state-space is instantiated by first selecting a grounded object set which consists of n objects $O = \{o_i\}_{i=1}^{n}$ where each $o \in O$ is an instance of some object class C. The value of attribute $C.\alpha$ for object o is denoted by $o.\alpha$. Then the grounded state-space, denoted $\mathcal{S}_O$, is an assignment of each $o.\alpha$ for all objects in O. The schema state-space, denoted $\mathcal{S}$, is

the set of all states for all possible object sets O.

To make the notion of a schema state-space concrete, consider the domain of Sokoban. Sokoban has four object classes: *Person*[2], *Box*, *Wall* and *Storage*. Each object class has attributes x and y for their location in the warehouse. Then any Sokoban state can be represented by instantiating objects of the four object classes and setting their x and y attributes accordingly. For example the game state in Figure 2.2a would be comprised of a state that contains $n = 46$ grounded objects. These would comprise of forty-one objects of class $Wall$, two objects of class *Box*, two objects of class *Storage* and one object of class *Person*. Each object would have their x and y attributes set accordingly to describe their x- and y-coordinates in the warehouse.

The RMDP framework focuses on the planning problem in RL. That is, the assumption is that the environment dynamics are known. Under this assumption, it is shown that for certain domains RMDPs can be used to learn generalised value functions that transfer between grounded MDPs instantiated from the schema [Guestrin et al., 2003, Guestrin, 2003].

4.3.3 The Propositional OO-MDP Formalism

The RMDP framework assumes known environment dynamics. In many cases of RL, the agent does not know these upfront and must learn them through experience. The Propositional OO-MDP formalism [Diuk et al., 2008, Diuk, 2010] introduces a framework that uses the same object-oriented schema state-space representation as RMDPs and proposes an algorithm to efficiently learn the transition dynamics, while assuming known reward dynamics.

In particular, Propositional OO-MDPs enforce a restriction that the transition dynamics of a domain must be described in terms of conjunctions over propositional statements that may only reference object classes. These conjunctions then map to effects over object class attributes. This means that the expressions used under Propositional OO-MDPs may only refer to lifted object classes, and not to grounded objects.[3] Under this restriction, Propositional OO-MDPs introduce a KWIK-learning algorithm to learn a model of the transition dynamics that can transfer across all MDPs instantiated from the schema. This was first done for the case of deterministic transition dynamics [Diuk et al., 2008], and later extended to a family of stochastic transition dynamics [Diuk, 2010].

To illustrate the Propositional OO-MDP formalism, consider the well-known Taxi

[2]We use *Person* instead of *Warehouse Keeper* for brevity.

[3]In principle, the Propositional OO-MDP formalism can utilise propositional statements over grounded objects. However, this complicates both learning and transfer procedures as discussed in Section 4.3.4.

task [Dietterich, 1998] as show in Figure 4.1. The original Taxi task takes place in a 5×5 gridworld where a taxi has to pick up a passenger that is at one of four pickup locations and drop them off at one of four destination locations. The possible pickup and destination locations as shown in Figure 4.1 and are fixed upfront for the task.

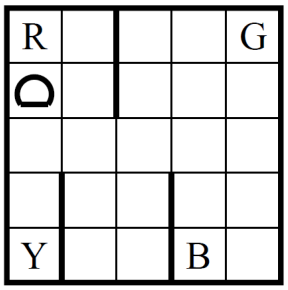


Figure 4.1: The original Taxi task (image source: [Dietterich, 1998]). Letters mark possible pickup and destination locations; marks the taxi; thicker lines mark walls.

The set of actions available to the agent are *North*, *East*, *South* and *West* that control the taxi's navigation in the gridworld, as well *Pickup* and *Dropoff*. The action *Pickup* picks up the passenger if the taxi is at the pickup location and the passenger is not already in the taxi, while the action *Dropoff* drops off the passenger if the taxi has already picked up the passenger and the taxi is on the destination location. Walls in the gridworld limit the taxi's movements. The reward dynamics for the Taxi task is -1 for each navigation step, -10 for applying the *Pickup* or *Dropoff* action incorrectly and 0 for reaching a terminal state that drops off the passenger at the correct destination location.

Due to the small state-space and action-space, the original Taxi task is easily solved using standard RL algorithms, like Q-learning, by representing the task as an MDP in a similar way to the Sokoban task described in Section 2.1.2. We can generalise the Taxi task to the Taxi domain which allows for Taxi tasks to be instantiated with different gridworld dimensions, wall locations, pickup locations and destination locations. An example of a more complex Taxi task in a 10×10 gridworld is shown in Figure 4.2.

To solve such larger Taxi tasks with standard Q-learning is prohibitive. However, object-oriented approaches provide a more sample efficient approach through transferable representations. Under the Propositional OO-MDP formalism, we would define the schema state-space with four object classes: *Taxi*, *Wall*, *Passenger* and *Destination*. Each object class has attributes x and y for their location in the grid. Object class *Wall* has an additional attribute *pos* to mark one of four positions in a square, while *Passenger* has an additional Boolean attributes *in-taxi* to indicate if the passenger is in the taxi. Given the schema state-space for this domain we

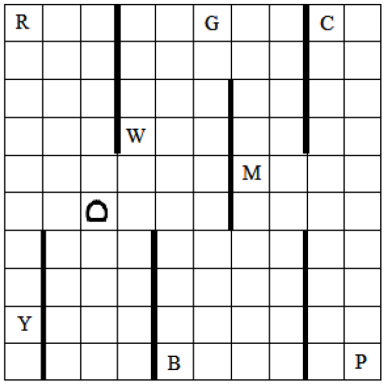


Figure 4.2: A 10×10 Taxi task from the Taxi domain with 8 possible pickup and destination locations (image source: [Diuk et al., 2008]). Letters mark possible pickup and destination locations; marks the taxi; thicker lines mark walls.

can instantiate a set of grounded objects from the object classes and a resulting grounded state.

As shown in table 4.1, the transition dynamics of the Taxi domain can then be described compactly with propositions over obejct classes. The statement

$$Touch_E(Taxi, Wall) = 0 \implies Taxi.x = Taxi.x + 1$$

for action *East* can be read as: if an object of class *Taxi* has an object of class *Wall* one square east of it is false, then all objects of class *Taxi* have their x attributes increased by 1. The same logic applies to the $Touch_N$, $Touch_S$ and $Touch_W$ expressions for the *North*, *South* and *West* actions respectively. The statement

$$On(Taxi, Passenger) = 1 \implies Passenger.in\text{-}taxi = 1$$

for can be read as as: if an object of class *Taxi* is on the same square as an object of class *Passenger* is true, then every object of class *Passenger* has its *in-taxi* attribute set to 1.

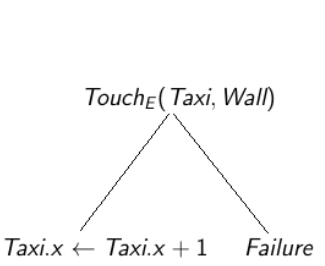
The important thing to note is that none of the conditions or effects in table 4.1 refer to any grounded object. This implies that this representation of the transition dynamics is fully lifted, and can therefore be used as the transition dynamics for the entire Taxi domain. Therefore, if we can learn it on some source tasks of the Taxi domain then we can transfer it to any target task of the Taxi domain without any additional learning and such transfer can greatly improve sample efficiently.

An alternative view of transition dynamics under the Propositional OO-MDP formalism is through binary trees. For each action a and attribute C_α, we can repre-

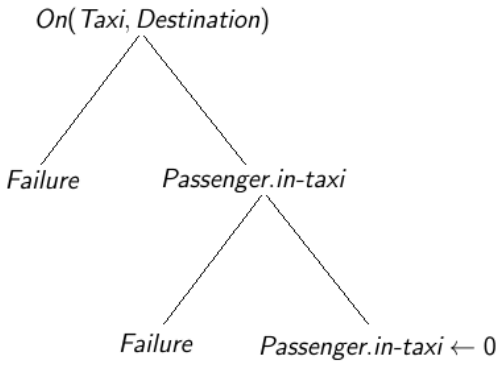
Action	Precondition	Effect
North	$Touch_N(Taxi, Wall) = 0$	$Taxi.y \leftarrow Taxi.y + 1$
East	$Touch_E(Taxi, Wall) = 0$	$Taxi.x \leftarrow Taxi.x + 1$
South	$Touch_S(Taxi, Wall) = 0$	$Taxi.y \leftarrow Taxi.y - 1$
West	$Touch_W(Taxi, Wall) = 0$	$Taxi.x \leftarrow Taxi.x - 1$
Pickup	$On(Taxi, Passenger) = 1$	$Passenger.in\text{-}taxi \leftarrow 1$
Dropoff	$Passenger.in\text{-}taxi = 1 \wedge On(Taxi, Destination) = 1$	$Passenger.in\text{-}taxi \leftarrow 0$

Table 4.1: Transition dynamics of the Taxi domain under the Propositional OO-MDP formalism. Under the formalism, any other assignment of truth values to the propositions leads to a global *failure condition* that leaves all object class attributes unchanged.

sent the transition dynamics as a full binary tree with propositions at the non-leaf nodes and effects at the leaf nodes. This is shown in Figure 4.3a for the action *East* and attribute *Taxi.x* and in Figure 4.3b for the action *Dropoff* and attribute *Passenger.in-taxi*.



(a) Full binary tree for the action *East* and attribute *Taxi.x*.



(b) Full binary tree for the action *Dropoff* and attribute *Passenger.in-taxi*.

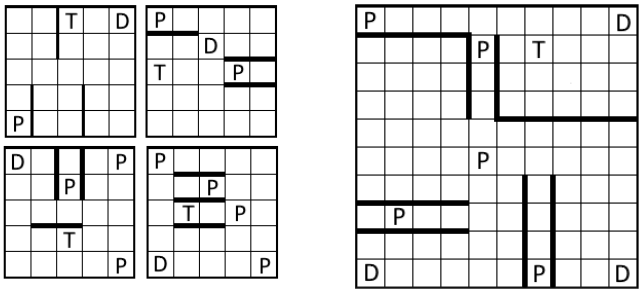
Figure 4.3: Examples of full binary tree structures under the Propositional OO-MDP formalism. The leaf node 'Failure' refers to a failure condition. Right branches represent a truth value of 1.

4.3.4 Limitations of Propositional OO-MDPs

The Propositional OO-MDP formalism is sufficient to represent the transition dynamics of the Taxi domain. However, the formalism's expressive power is limited because it is restricted to propositional statements. In particular, Propositional OO-MDPs cannot compactly represent the transition dynamics of domains where it is required to distinguish between different objects of the same object class.

To illustrate this, consider a simple extension to the Taxi domain which we call the

All-Passenger Any-Destination Taxi domain. This domain is similar to the original Taxi domain except that the taxi is tasked to pick up multiple passengers and drop each of them off at one of any destination locations. The taxi can only pick up one passenger at a time, so if a passenger is already in the taxi and the *Pickup* action is taken while the taxi is at the pickup location of another passenger, the state does not change. The state-space of the schema for this domain under an object-oriented representation is identical to that of the original Taxi domain described in Section 4.3.3 except that we add an additional Boolean attribute to the *Passenger* object class *at-destination*, to indicate if a passenger has already been dropped off at a destination. Figure 4.4 shows sample states of the schema.



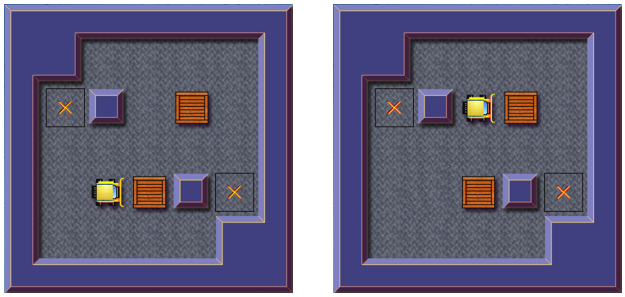
(a) (b)

Figure 4.4: 'P' marks a passenger; 'D' marks a destination; 'T' marks a taxi; thicker lines mark walls. Five possible states from the All-Passenger Any-Destination Taxi domain schema.

Propositions over object classes are insufficient to compactly represent the transition dynamics for this version of the Taxi domain. To see why, suppose we have a task with two passenger objects and the proposition $On(Taxi, Passenger)$ with a truth value 1. This can be read as: an object of class *Taxi* is on the same square as an object of class *Passenger* is true. Clearly, this information is insufficient to determine which passenger object's *in-taxi* attribute should change given the *Pickup* action. To overcome this ambiguity under a propositional approach, we must resort to propositions over the grounded passenger objects, $passenger_1$ and $passenger_2$, of the form $On(Taxi, passenger_1)$ and $On(Taxi, passenger_2)$. Note that while this resolves the ambiguity, the number of propositions needed for the precondition now changes as we change the number of *passenger* objects in the task. This complicates both learning and transfer procedures.

As a further example, consider the Sokoban domain with the propositions $Touch_W(Box, Person)$ and $Touch_E(Box, Wall)$ both with truth value 1. The first proposition can be read as: an object of class *Box* has an object of class *Person* one square west of it is true, while the second can be read as: an object of class *Box* has an object of class *Wall* one square east of it is true. Then the conjunction

$Touch_W(Box, Person) = 1 \wedge Touch_E(Box, Wall) = 1$ is insufficient to determine the transition dynamics of any *box* object's x attribute when taking action *East* since there is no way to know if the statements are referring to the same box. See Figure 4.5 for an illustration. A similar example of ambiguity that is not resolvable under a proportional approach is also described by the original authors of the Propositional OO-MDP framework [Diuk, 2010].



(a) (b)

Figure 4.5: For action *East* and the box adjacent to the warehouse keeper, in figure (4.5a) the effect is $box.x \leftarrow box.x + 0$; in figure (4.5b) the effect is $box.x \leftarrow box.x + 1$ while the conjunction $Touch_W(Box, Person) = 1 \wedge Touch_E(Box, Wall) = 1$ is true in both cases.

4.3.5 Deterministic Object-Oriented R_{max}

While Propositional OO-MDPs have limited expressive power, it has been shown that under this formalism efficient learning is possible under the KWIK framework. In particular, the authors who introduced the Propositional OO-MDP formalism also introduced an accompanying algorithm called Deterministic Object-Oriented R_{max} (*DOORMAX*) that can be used to KWIK-learn the transition dynamics for deterministic environments that are representable under the formalism [Diuk et al., 2008, Diuk, 2010]. We outline the core idea of the algorithm in this section.[4]

The *DOORMAX* algorithm makes the following main assumptions. For each action a and attribute C_α:

- There is one true effect type, *eff*. An effect type describes the possible effects that can be applied to C_α when taking action a. For example, if C_α is an integer attribute, the relative effect type is $Rel_i(x) = x + i$ where $i \in \mathbb{Z}$ while the absolute effect type is $Abs(x) = c$ where $c \in \mathbb{Z}$ is some constant.
- There is a pre-specified set of propositions, and all observed effects in *eff* can be described by taking conjunctions over a subset of these propositions with truth values assigned.

[4] In Section 4.6 we introduce an algorithm called $DOORMAX_D$ which is analogous to *DOORMAX*. Therefore, we only outline the original algorithm *DOORMAX* in this section.

- The set of conjunctions that map to effects under an effect type *eff* must induce a full-binary tree with propositions at the leaf-nodes and effects at the non-leaf nodes. Each effect in *eff* can occur at most at one non-leaf node in the tree. All leaf nodes that do not map a unique effect in *eff* map to a failure condition.

As an illustrative example for the original Taxi domain, consider the action *East*, attribute *Taxi.x* and effect type Rel_i. We can propose four propositional statements $Touch_N(Taxi, Wall)$, $Touch_E(Taxi, Wall)$, $Touch_S(Taxi, Wall)$ and $Touch_W(Taxi, Wall)$. The aim of the KWIK-learner then is to learn that

$$Touch_E(Taxi, Wall) = 0 \implies Taxi.x \leftarrow Taxi.x + 1$$

when action *East* is taken.

The key insight that underlies the *DOORMAX* algorithm is that conjunctions that map to unique effects can be efficiently learned because we can potentially invalidate multiple propositions with a single observation. To illustrate this, suppose that agent interacts with its environment and takes action *East* to observe the effect: *Taxi.x*+1. Prior to taking the action *East*, the agent's state determined the following assignments to the propositions:

$Touch_N(Taxi, Wall) = 0 \wedge Touch_E(Taxi, Wall) = 0 \wedge Touch_S(Taxi, Wall) = 0 \wedge Touch_W(Taxi, Wall) = 0$.

Suppose the next time the agent takes the action *East* it observes the effect *Taxi.x*+1 and the agent's previous state determined the following assignments to the propositions:

$Touch_N(Taxi, Wall) = 1 \wedge Touch_E(Taxi, Wall) = 0 \wedge Touch_S(Taxi, Wall) = 1 \wedge Touch_W(Taxi, Wall) = 1$.

Since the *DOORMAX* algorithm assumes that each effect can occur at most once in the tree, we can infer that the propositions $Touch_N(Taxi, Wall)$, $Touch_S(Taxi, Wall)$ and $Touch_W(Taxi, Wall)$ are irrelevant to the precondition of the effect. As a result the agent has learned

$$Touch_E(Taxi, Wall) = 0 \implies Taxi \leftarrow Taxi.x + 1$$

with only two unique observations. Of course, this is a best-case scenario. A worst-case scenario requires 4 unique observations. In general, given $D \geq 0$ propositions, learning requires at most $D + 1$ unique observations.[5]

[5]It is $D + 1$ and not D because an effect can depend on no propositions.

If we do not assume that conjunctions map to unique effects, such efficient learning is not possible. Instead, the agent must observe every possible truth value assignment to the propositional statements and map each one to the corresponding effect. This requires 2^D unique observations which is prohibitive if D is large.

The Propositional OO-MDP framework extends to allow for the learner to hypothesise $N > 0$ effect types under the assumption that exactly one effect type is true and, furthermore, that the maximum number of effects per effect type is bounded by some $K > 0$. Under these additional assumptions one can prove that *DOORMAX* has a KIWK-bound $K(D+1)+1$ to learn a single effect type for an action a and attribute C_α, while learning given N effect types has a KWIK-bound of $N(K(D+1)+1)+(N-1)$ [Diuk et al., 2008]. These bounds exclude the learning of failure conditions. Learning failure conditions is done through memorisation on a case by case basis. For example, once it is observed that for the action *East* the precondition

$Touch_N(Taxi, Wall) = 0 \wedge Touch_E(Taxi, Wall) = 1 \wedge Touch_S(Taxi, Wall) = 0 \wedge Touch_W(Taxi, Wall) = 0$

leads to an unchanged state, then this precondition is memorised for future use. Note that learning failure conditions is inefficient because each failure condition must be recorded individually.

4.4 The Deictic Object-Oriented Framework

The core restriction of Propositional OO-MDPs that the preconditions be described only in terms of propositions tends to be a strong one and, as discussed in Section 4.3.4, precludes the compact representation of certain domains. Such domains include those where tasks from the domain are required to distinguish between different objects of the same object class.

In this chapter we propose to overcome the limitations of Propositional OO-MDPs with a novel deictic object-oriented formalism, Deictic OO-MDPs. The key insight behind Deictic OO-MDPs is the concept of a deictic predicate. A deictic predicate is a predicate that is grounded only with respect to a central deictic object that relates itself to non-grounded object classes. Returning to the Sokoban domain, a deictic predicate over boxes allows a specific box to ascertain whether *any* wall is adjacent to it, but not whether a *specific* wall is adjacent to it.

In this section we present the Deictic OO-MDP framework. In Section 4.4.1 we present the formalism under a schema; in Section 4.4.2 we demonstrate that the framework is able to resolve the limitations of Propositional OO-MDPs; and in

Section 4.4.3 we present an algorithm to efficiently learn deterministic transition dynamics under this formalism.

4.4.1 Formalism

The Deictic OO-MDP framework uses the same schema state-space $\mathcal{S}$ as described in Section 4.3.2 while deictic predicate preconditions are used to define the schema transition dynamics as described below. Let $\mathcal{A}$ be a set of actions. Then for each action $a \in \mathcal{A}$ and attribute $C.\alpha$ of a class C define a set of effects of size $K_{a,C.\alpha}$:

$$\mathcal{E}_{a,C.\alpha} = \{e_i : Dom(C.\alpha) \rightarrow Dom(C.\alpha)\}_{i=1}^{K_{a,C.\alpha}},$$

Define a set of deictic predicates of size $D_{a,C.\alpha}$:

$$\mathcal{F}_{a,C.\alpha} = \{f_i : O[C] \times S \rightarrow \mathfrak{B}\}_{i=1}^{D_{a,C.\alpha}},$$

where $O[C]$ is a set that contains objects with all possible attribute value assignments that are instances of C, and $\mathfrak{B} = \{0, 1\}$.

Then the probabilistic transition dynamics for a and $C.\alpha$ are defined by

$$\mathcal{P}_{a,C.\alpha} : \mathfrak{B}^{D_{a,C.\alpha}} \times \mathcal{E}_{a,C.\alpha} \rightarrow [0, 1].$$

The schema transition dynamics $\mathcal{P}$ is the set of transition dynamics for all actions and attributes,

$$\mathcal{P} = \{\mathcal{P}_{a,C.\alpha} | a \in \mathcal{A}, C \in \mathfrak{C}, C.\alpha \in Att(C)\}.$$

The schema reward dynamics are defined by

$$\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}.$$

Given an object set O we can instantiate a grounded MDP $\mathcal{M}_O = (\mathcal{S}_O, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. Then if the agent is currently in state $s \in \mathcal{S}_O$ and takes action a, the transition dynamics for $\mathcal{M}_O$ operate as follows: for each object o in s that is an instance of C and each attribute $C.\alpha$ we compute the Boolean truth values $\mathcal{B} = \{f_i(o, s)\}_{i=1}^{D_{a,C.\alpha}}$ for the deictic predicates in $\mathcal{F}_{a,C.\alpha}$. Then for an effect $e \in \mathcal{E}_{a,C.\alpha}$ we compute $\mathcal{P}_{a,C.\alpha}(\mathcal{B}, e)$ which returns the probability of e occurring given $\mathcal{B}$. This implies a distribution

over effects which in turn implies a distribution over the attribute values of o by applying the effect to $o.\alpha$ in s and obtaining $e(o.\alpha) = o.\alpha'$ in s'.

As an example, consider attribute *Taxi.x* and action *East* for the Taxi domain described in Section 4.3.3. We can define a set of relative effects $Rel_i(x) = x + i$ that produce a shift of i squares from the current location x, as well as a deictic predicate $Touch_E(taxi, s)$ that returns 1 if the *taxi* object has an object of class *Wall* one square east of it in s, otherwise 0. Then the transition dynamics can be described for any *taxi* object as

$$Touch_E(taxi, s) = 1 \implies taxi.x \leftarrow Rel_0(taxi.x)$$

with probability one and

$$Touch_E(taxi, s) = 0 \implies taxi.x \leftarrow Rel_1(taxi.x)$$

with probability one.

The key insight with Deictic OO-MDPs is that the parameters we pass to each deictic predicate in $\mathcal{F}_{a,C.\alpha}$ are a grounded deictic object o that must be an instance of C and s which is a state *of the schema*, not a grounded state. As a result, these deictic predicates may not refer to specific objects in s; however, they may relate o to object classes of the schema. For example, with $Touch_E(taxi, s)$ as defined above only *taxi* is grounded while we never refer to a grounded object in s.

Given a resulting state s', the reward dynamics operate by computing $\mathcal{R}(s, a, s')$. For the purposes of this section we will assume that $\mathcal{R}$ is known while $\mathcal{P}$ needs to be learned. We present a formalism that allows for efficient learning of $\mathcal{R}$ in Section 4.5.

Similarly to the Propositional OO-MDP formalism, a requirement for the definition of $\mathcal{P}_{a,C.\alpha}$ to be correct is that the set $\mathcal{E}_{a,C.\alpha}$ must be *invertible* so that if the current value assignment for some attribute $C.\alpha$ of a grounded object o in state s is $o.\alpha$ and we take action a to subsequently observe $o.\alpha'$ in s' then there must be a unique $e \in \mathcal{E}_{a,C.\alpha}$ such that $e(o.\alpha) = o.\alpha'$. Clearly, if this requirement is not met then either the effects are not able to correctly capture the true transition dynamics or there are duplicate effects that lead to the same $o.\alpha'$ which creates ambiguity over which effect occurred.

Note that the Deictic OO-MDP formalism is more general than the Propositional OO-MDP formalism since a deictic predicate that does not make reference to its de-

ictic object is simply a proposition. Consequently, any domain that is representable under the Propositional OO-MDP formalism is also representable under the Deictic OO-MDP formalism.

4.4.2 Resolving the Limitations of Propositional OO-MDPs

In Section 4.3.4 we demonstrate that Propositional OO-MDPs are insufficiently expressive to represent the transition dynamics of the All-Passenger Any-Destination Taxi and Sokoban domains. Deictic OO-MDPs are more expressive allowing us to compactly represent the transition dynamics for both these domains.

Consider the action *Pickup* and Boolean attribute *Passenger.in-taxi* in the All-Passenger Any-Destination Taxi domain with the effect $SetBool_i(x) = x\mathbb{1}(i = 0) + (1 - x)\mathbb{1}(i = 1)$ for $i \in \{0, 1\}$ where $\mathbb{1}$ is the indicator function. The effect function $SetBool_i$ flips a Boolean x is $i = 1$ and leaves it unchanged if $i = 0$. Under a propositional approach we cannot distinguish between the different passenger objects to determine which passenger's *in-taxi* attribute should be set to 1. However, with a deictic representation we can resolve this ambiguity. Specifically, given a state s of the schema we can define two deictic predicates relative to a *passenger*, as well as one propositon:

- $AnyTaxiOnPassenger(passenger, s)$ that returns a truth value of 1 if any object of class *Taxi* is on the same square as *passenger* in s, and 0 otherwise.
- $PassengerAtAnyDestination(passenger, s)$ that returns a truth value of 1 if *passenger.at-destination* is set to 1 in s, and 0 otherwise.
- $AnyPassengerInAnyTaxi(passenger, s)$ that returns a truth value of 1 if there is any object of class *Passenger* that has its *in-taxi* attribute set to 1 in s, and 0 otherwise.[6]

Then if

$AnyTaxiOnPassenger(passenger, s) = 1 \wedge PassengerAtAnyDestination(passenger, s) = 0 \wedge AnyPassengerInAnyTaxi(passenger, s) = 0,$

we have that

$$passenger.in\text{-}taxi \leftarrow SetBool_1(passenger.in\text{-}taxi),$$

[6]The statement $AnyPassengerInAnyTaxi(passenger, s)$ is actually a proposition as it does not refer to the grounded *passenger* object. We could simplify notation by writing $AnyPassengerInAnyTaxi(s)$. However, we include the *passenger* object to remain consistent with the notation described in Section 4.4.1.

and otherwise

$$passenger.in\text{-}taxi \leftarrow SetBool_0(passenger.in\text{-}taxi).$$

Deictic OO-MDPs are also able to represent the transition dynamics of the Sokoban domain without ambiguity. Consider Figure 4.5 where under the Propositional OO-MDP formalism we have ambiguity when taking the action *East* as to which box object's x attribute should change. We can resolve this ambiguity under the Deictic OO-MDP framework by defining the following deictic predicates relative to a grounded *box* object:

- $AnyPersonWestOfBox(box, s)$ that returns a truth value of 1 if *box* has any object of class *Person* one square west of it in s, otherwise 0.
- $AnyWallEastOfBox(box, s)$ that returns a truth value 1 if *box* has any object of class *Wall* one square east of it in s, otherwise 0.

Then

$AnyPersonWestOfBox(box, s) = 1 \wedge AnyWallEastOfBox(box, s) = 1 \implies box.x \leftarrow Rel_0(box.x)$.

In fact, representing the transition dynamics for this domain in terms of Deictic OO-MDPs is straightforward. For a deictic *person* object there are four deictic predicates required when taking the action *East* that resolve the following questions: is there any object of class *Box* one square east of *person*? Is there any object of class *Box* two squares east of *person*? Is there any object of class *Wall* one square east of *person*? Is there any object of class *Wall* two squares east of *person*? Meanwhile for a deictic *box* object there are three deictic predicates required to resolve the questions: is there any object of class *Person* one square west of *box*? Is there any object of class *Wall* one square east of *box*? Is there any object of class *Box* one square east of *box*? The other actions are analogous.

See Appendix Sections B.1.1 and B.2.1 for more details on the transition dynamics under the Deictic OO-MDP formalism for the All-Passenger Any-Destination Taxi and Sokoban domains respectively.

4.4.3 Algorithms

Given a set of D deictic predicates we want to learn the transition dynamics for each action a and attribute $C.\alpha$. If the transition dynamics are deterministic this can be done using memorisation with 2^D unique observations. However, this is prohibitive if D is large. As discussed in Section 4.3.5 Propositional OO-MDPs introduce a

learning algorithm called *DOORMAX* for deterministic transition dynamics that, under certain assumptions, has a KWIK-bound that is linear in D.

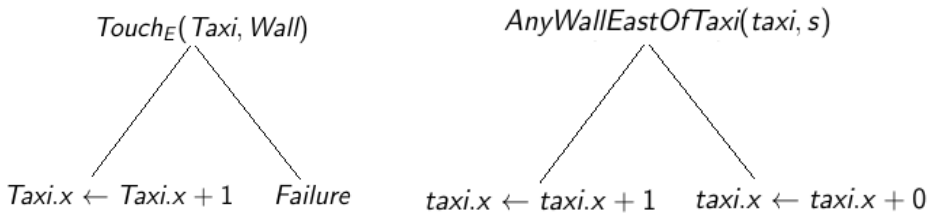
For *DOORMAX* to be correct, the transition dynamics for each action and attribute must be representable as a full binary tree with propositions at the non-leaf nodes and effects at the leaf nodes. Furthermore, each possible effect of an effect type can occur at most at one leaf node of the tree, except for a special effect called a *failure condition* that may occur at multiple leaf nodes. A failure condition implies that globally no attribute changes when an action was taken i.e. $s = s'$ when a is taken. See Figure 4.6a for how this represented for the *Taxi.x* attribute with action *East*.

The intuition behind *DOORMAX* is that, in many cases, the number of propositions that an effect depends on is much smaller than D. Furthermore, since an effect can occur at most once in the tree, we can invalidate multiple propositions with a single observation. We adapt the *DOORMAX* algorithm to Deictic OO-MDPs, which we call $DOORMAX_D$ (Algorithm 13). The $DOORMAX_D$ algorithm requires two sub-algorithms to learn and make predictions for the transition dynamics, and these sub-algorithms are presented in this section. The full $DOORMAX_D$ algorithm that calls these sub-algorithms is then presented in Section 4.6.

The main difference between $DOORMAX_D$ and *DOORMAX* is that we remove the notion of a global failure condition. Instead we require that all effects apply to a single attribute. See Figure 4.6b for how this is represented for the *Taxi.x* attribute with action *East*. To achieve this, we extend the notion of an effect type to include a partition function over effects that groups them into those that can occur at most at one leaf node and those that can occur at multiple leaf nodes. For example, as shown in figure 4.6*b*, the effects $taxi.x \leftarrow taxi.x + 1$ (Rel_1) and $taxi.x \leftarrow taxi.x + 0$ (Rel_0) are both unique in the tree. Therefore, by using an appropriate partition function, we can efficiently learn both these branches. Meanwhile, the design choice used by *DOORMAX* of a global failure conditions implies that $Taxi.x \leftarrow Taxi.x + 1$ (Rel_1) is efficiently learned, while $Taxi.x \leftarrow Taxi.x + 0$ (Rel_0) is learned by memorisation. A trade-off exists between these design choices. The $DOORMAX_D$ algorithm requires a partition function for each action, attribute and effect set that improves learning efficiency. However, constructing such a partition function requires additional prior knowledge about a domain that is not required by *DOORMAX*.

In this section we introduce two algorithms that are called by $DOORMAX_D$ to KWIK-learn the transition dynamics of an MDP under the Deictic OO-MDP formalism:

- *UpdateTree1* (Algorithm 7) updates the binary tree for action a and attribute $C.\alpha$.



(a) Full binary tree under Propositional OO-MDPs.

(b) Full binary tree under Deictic OO-MDPs.

Figure 4.6: Full binary tree structure for the transition dynamics of $Taxi.x$ attribute and action *East*. Right branches represent a truth value of 1.

- *Predict1* (Algorithm 8) makes a prediction for action a and attribute $C.\alpha$.

Before introducing the algorithms, we require some definitions. Let $\mathcal{F}$ be a set of deictic predicates and $\mathcal{E}$ be a set of effects.

Definition 5. A term is a tuple (f, b) where $f \in \mathcal{F}$ and $b \in \mathfrak{B}$. A set of terms is denoted by $\mathcal{T}$. A set that contains sets of terms is denoted by $\mathfrak{T}$.

Definition 6. Term (f_1, b_1) mismatches term (f_2, b_2) if $f_1 = f_2$ and $b_1 \neq b_2$.

Definition 7. $\Pi : \mathcal{E} \rightarrow \mathfrak{B}$ is a binary partition function over $\mathcal{E}$ and assigns each effect in $\mathcal{E}$ to one of two partitions, 0 or 1. We call the tuple $g = (\mathcal{E}, \Pi)$ an effect type. Denote by K_0^g and K_1^g the number of effects in partition 0 and 1 respectively. We use . notation to refer to an element in a tuple g, so for example $g.\mathcal{E}$ refers to $\mathcal{E}$ in g. We denote sets of effect types with $\mathcal{G}$.

Definition 8. Let g be an effect type. Let $M > 1$ be a constant. Then $Tree(g, \mathcal{F}, M)$ is the set of all full binary trees such that non-leaf nodes are elements of $\mathcal{F}$ and leaf nodes are elements of $g.\mathcal{E}$. Furthermore, if $g.\Pi$ assigns an effect in $g.E$ to partition 1 then that effect can occur at most at one leaf node and we call that effect conjunctive, otherwise that effect may occur at most at M leaf nodes and we call that effect disjunctive.

We now introduce the *UpdateTree1* (Algorithm 7) and *Predict1* (Algorithm 8) algorithms. Note that $\hat{\mathcal{F}}(a, C.\alpha)$, $\hat{\mathcal{G}}(a, C.\alpha)$, $\mathfrak{T}_e^g(a, C.\alpha)$ and M are initialised globally in $DOORMAX_D$ as discussed in Section 4.6. The set $\hat{\mathcal{F}}(a, C.\alpha)$ contains hypothesised deictic predicates for action a and attribute $C.\alpha$; the set $\hat{\mathcal{G}}(a, C.\alpha)$ contains hypothesised effect types for action a and attribute $C.\alpha$; while the set $\mathfrak{T}_e^g(a, C.\alpha)$ contains the sets of terms that map to effect e of effect type g for action a and attribute $C.\alpha$. The $\hat{}$ notation used for $\hat{\mathcal{F}}(a, C.\alpha)$ and $\hat{\mathcal{G}}(a, C.\alpha)$ indicates that these sets are sets of hypotheses, of which the correct hypotheses must be determined by the algorithm.

At a high-level, the Algorithm *UpdateTree1* ensures that for all $g \in \hat{\mathcal{G}}(a, C.\alpha)$ the

Algorithm 7: *UpdateTree1*: update binary tree for action a and attribute $C.\alpha$.

Input: $C.\alpha \in Att(C)$, $s \in \mathcal{S}$, $o \in O[C]$, $a \in \mathcal{A}$, $o.\alpha' \in Dom(C.\alpha)$

pass o and s to the deictic predicates in $\hat{\mathcal{F}}(a, C.\alpha)$ to retrieve a set of terms $\mathcal{T}$
foreach $g \in \hat{\mathcal{G}}(a, C.\alpha)$ **do**
  foreach $e \in g.\mathcal{E}$ **do**
    if $e(o.\alpha) = o.\alpha'$ **then**
      if $\mathfrak{T}_e^g(a, C.\alpha) = \phi$ **then**
        if $\exists e' \in g.\mathcal{E}$ *with* $\mathcal{T}_{e'} \in \mathfrak{T}_{e'}^g(a, C.\alpha)$ *such that* $\mathcal{T}_{e'} \subseteq \mathcal{T}$ **then**
          remove g from $\hat{\mathcal{G}}(a, C.\alpha)$
        else
          add $\mathcal{T}$ to $\mathfrak{T}_e^g(a, C.\alpha)$
        end
      else
        if $g.\Pi(e) = 0$ **then**
          add $\mathcal{T}$ to $\mathfrak{T}_e^g(a, C.\alpha)$
        else
          $\mathcal{T}_{temp} \leftarrow \mathcal{T}$
          $\mathcal{T} \leftarrow$ the only element in $\mathfrak{T}_e^g$
          remove from $\mathcal{T}$ any terms that mismatch terms in $\mathcal{T}_{temp}$
          $\mathfrak{T}_e^g(a, C.\alpha) \leftarrow \phi$
          add $\mathcal{T}$ to $\mathfrak{T}_e^g(a, C.\alpha)$
        end
        if $(\exists e' \in (g.\mathcal{E} - \{e\})$ *with* $\mathcal{T}_{e'} \in \mathfrak{T}_{e'}^g(a, C.\alpha)$ *such that* $(\mathcal{T} \subseteq \mathcal{T}_{e'}$ *or* $\mathcal{T}_{e'} \subseteq \mathcal{T}))$ *or* $|\mathfrak{T}_e^g(a, C.\alpha)| > M$ **then**
          remove g from $\hat{\mathcal{G}}(a, C.\alpha)$
        end
      end
    end
  end
end

constraint $\mathit{Tree}(g, \mathcal{F}(a, C.\alpha), M)$ is maintained. That is, the set $\{\mathfrak{T}_e^g(a, C.\alpha)\}_{e \in g.\mathcal{E}}$ induces a binary tree subject to the constraints of the partition function $g.\Pi$. Each time we observe a set of terms $\mathcal{T}$ and an associated effect e we update $\mathfrak{T}_e^g(a, C.\alpha)$, and in doing so may discover that the resulting set $\{\mathfrak{T}_e^g(a, C.\alpha)\}_{e \in g.\mathcal{E}}$ can no longer induce an appropriate binary tree at which point we remove g from $\hat{\mathcal{G}}(a, C.\alpha)$.

At the lower-level, the Algorithm *UpdateTree1* updates the binary tree for action a and attribute $C.\alpha$ given an object o from state s and the resulting attribute value $o.\alpha'$ in s'. The algorithm starts at line 1 where it computes $\mathcal{T}$, the terms for $\hat{\mathcal{F}}(a, C.\alpha)$. In lines 2-4 the algorithm iterates over each effect type $g \in \hat{\mathcal{G}}(a, C.\alpha)$ and each effect $e \in g.\mathcal{E}$ where it checks if e applied to $o.\alpha$ is equal to $o.\alpha'$. If it is, then it proceeds to update $\mathfrak{T}_e^g(a, C.\alpha)$ in lines 5-25.

If $\mathfrak{T}_e^g(a, C.\alpha)$ is empty, then as per lines 5-11, the algorithm checks if $\mathcal{T}$ already maps to different effect in g. If it does, then the effect type g is invalidated because adding $\mathcal{T}$ to $\mathfrak{T}_e^g(a, C.\alpha)$ would result in an invalid tree; otherwise, $\mathcal{T}$ is added to $\mathfrak{T}_e^g(a, C.\alpha)$.

If $\mathfrak{T}_e^g(a, C.\alpha)$ is not empty then, as per lines 12-20, it first checks if e is disjunctive or conjunctive based on the partition function $g.\Pi$. If e is disjunctive then it adds $\mathcal{T}$ to $\mathfrak{T}_e^g(a, C.\alpha)$. If e is conjunctive then it updates the existing element in $\mathfrak{T}_e^g(a, C.\alpha)$ by removing from it any terms that mismatch terms in $\mathcal{T}$ - this refines the set of terms that e depends on in the tree, and is the core mechanism for efficient learning in $\mathit{DOORMAX}_D$.

Finally, after the update to $\mathfrak{T}_e^g(a, C.\alpha)$, in lines 21-23 the algorithm checks if $\mathcal{T}$ maps to a different existing effect in g or if the number of effects in $\mathfrak{T}_e^g(a, C.\alpha)$ exceeds the limit M. In either of these cases, g is invalidated.

To provide a better intuition on the invalidation logic of the algorithm, consider a case where $\mathcal{F} = \{f_1, f_2, f_3\}$ and there are two effects $\mathcal{E} = \{e_1, e_2\}$ where e_1 is conjunctive and e_2 is disjunctive. Consider the following examples:

- Figure 4.7a: we currently have $\mathfrak{T}_{e_1} = \{\mathcal{T}_{e_1} = \{(f_1, 1), (f_2, 1), (f_3, 1)\}\}$ and $\mathfrak{T}_{e_2} = \phi$. Suppose we then observe e_2 with $\mathcal{T} = \{(f_1, 1), (f_2, 1), (f_3, 1)\}$. Then $\mathfrak{T}_{e_2}$ is empty and $\mathcal{T}_{e_1} \subseteq \mathcal{T}$.
- Figure 4.7b: we currently have $\mathfrak{T}_{e_1} = \{\mathcal{T}_{e_1} = \{(f_1, 1), (f_2, 1), (f_3, 1)\}\}$ and $\mathfrak{T}_{e_2} = \{\mathcal{T}_{e_2} = \{(f_1, 1), (f_2, 1), (f_3, 0)\}\}$. Suppose we then observe e_1 with $\mathcal{T} = \{(f_1, 1), (f_2, 0), (f_3, 0)\}$. As e_1 is conjunctive and $\mathfrak{T}_{e_1}$ is not empty we first remove mismatching terms. Then $\mathfrak{T}_{e_1} = \{\mathcal{T} = \{(f_1, 1)\}\}$ and now $\mathcal{T} \subseteq \mathcal{T}_{e_2}$.
- Figure 4.7c: we currently have $\mathfrak{T}_{e_1} = \{\mathcal{T}_{e_1} = \{(f_1, 1)\}\}$ and $\mathfrak{T}_{e_2} = \{\mathcal{T}_{e_2} = \{(f_1, 0), (f_2, 0), (f_3, 0)\}\}$. Suppose we then observe e_2 with $\mathcal{T} = \{(f_1, 1), (f_2, 0), (f_3, 1)\}$. We add $\mathcal{T}$ to $\mathfrak{T}_{e_2}$ and now $\mathcal{T}_{e_1} \subseteq \mathcal{T}$.

In all the above cases, we conclude that the effect type is invalid since the observed data can no longer induce a binary tree subject to the specified constraints. Note that we do not place any restrictions on the order in which the preconditions may appear in the tree, but there is no reordering that can recover an appropriate binary tree given the data.

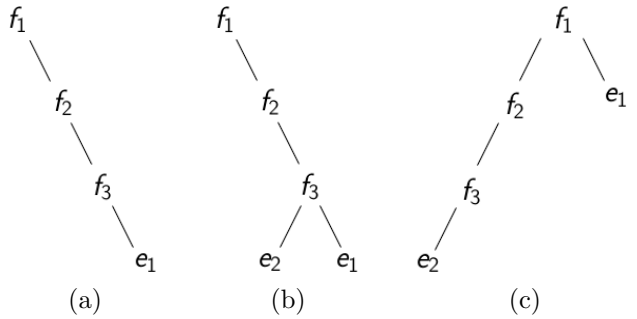


Figure 4.7: Binary trees induced by $\mathfrak{T}_{e_1}$ and $\mathfrak{T}_{e_2}$. Right branches represent a truth value of 1.

Algorithm 8: *Predict1*: prediction procedure for action a and attribute $C.\alpha$.

Input: $C.\alpha \in Att(C)$, $s \in S$, $o \in O[C]$, $a \in \mathcal{A}$

initialise an empty set $\mathcal{V}\langle Dom(C.\alpha)\rangle$
pass o and s to the deictic predicates in $\hat{\mathcal{F}}(a, C.\alpha)$ to retrieve a set of terms $\mathcal{T}$
foreach $g \in \hat{\mathcal{G}}(a, C.\alpha)$ **do**
    $pred \leftarrow \perp$
    foreach $e \in g.\mathcal{E}$ **do**
        if $\exists \mathcal{T}_e \in \mathfrak{T}_e^g(a, C.\alpha)$ *such that* $\mathcal{T}_e \subseteq \mathcal{T}$ **then**
            $pred \leftarrow e(o.\alpha)$
            exit loop
        end
    end
    if $pred = \perp$ **then**
        return $\perp$
    else
        add $pred$ to $\mathcal{V}$
        if $|\mathcal{V}| > 1$ **then**
            return $\perp$
        end
    end
end
return only element in $\mathcal{V}$

The *Predict1* procedure is analogous to that used in *DOOMAX*. At a high-level the algorithm makes a prediction for attribute $C.\alpha$ of an object o of class C given a schema state s and action a. The algorithm requires that for each effect type $g \in \hat{\mathcal{G}}(a, C.\alpha)$ a prediction is made that is not $\perp$, and furthermore that all effect types make the same prediction; otherwise the algorithm returns $\perp$. This ensures

that the algorithm only makes a correct prediction if it does not return $\perp$. Once the transition dynamics are learned, then given a state s and action a the resulting state s' can be determined by calling *Predict1* for each object attribute $o.\alpha$ in s to obtain $o.\alpha'$ in s'.

In line 2 of the algorithm it computes $\mathcal{T}$, the terms for $\hat{\mathcal{F}}(a, C.\alpha)$. In line 3 the algorithm iterates over the effect types $g \in \hat{\mathcal{G}}(a, C.\alpha)$. In lines 4-10 the algorithm checks whether any effect $e \in g.\mathcal{E}$ is mapped to by $\mathcal{T}$ in $\mathfrak{T}_e^g(a, C.\alpha)$. If it is, then the prediction $e(o.\alpha)$ is recorded and added to a set $\mathcal{V}$ in line 14; otherwise the algorithm returns $\perp$ in line 12. In lines 15-17 the algorithm checks if $|\mathcal{V}| > 1$. If it is, then there are two effect types that make different predictions for $o.\alpha$ and the algorithm returns $\perp$; otherwise, all effect types record the same prediction for $o.\alpha$ and the algorithm returns this prediction in line 20.

4.5 Learning Reward Dynamics under Propositional OO-MDPs

Propositional OO-MDPs introduce a KWIK-learning algorithm to learn the transition dynamics of a domain, while assuming known reward dynamics. In this section we introduce an algorithm to KWIK-learn a family of reward dynamics under a propositional object-oriented approach.

This family consists of reward dynamics whereby for most transitions the agent receives a default reward signal, while a small number of transition groups lead to different reward signals. An example of a domain that exhibits such reward dynamics is the Taxi domain. In the Taxi domain, the agent receives a default reward of -1 for all transitions except for the groups of transitions that: apply an illegal pickup operation, which produce a reward signal of -10; apply an illegal dropoff operation, which produce a reward signal of -10; or lead to terminal state, which produces a reward signal of 0. Furthermore, it is required that these groups can be described under an object-oriented approach through propositional statements over object class attributes.

We note that while it is possible to extend the propositional formalism described in this section to a deictic formalism, the structure of reward dynamics in most MDPs makes a deictic approach unnecessary. This is because most RL tasks have global goals such as 'get all boxes to some storage location' in the Sokoban domain or 'get all passengers to some destination location' in the All-Passenger Any-Destination Taxi domain.

4.5.1 Refactoring Reward Dynamics

We note that for any finite MDP, the reward dynamics can be refactored as a sum of $L + 1$ terms:

$$\mathcal{R}(s, a, s') = \sum_{i=1}^{L} z_i(s, a, s')U_i + (1 - \sum_{i=1}^{L} z_i(s, a, s'))U_{L+1}, \tag{4.1}$$

where the $z_i(s, a, s') \in \mathfrak{B}$ are indicator variables such that $\sum_{i=1}^{L} z_i(s, a, s') \in \mathfrak{B}$ - that is, for any (s, a, s') at most one z_i has value one and all others have value zero - and U_i is a *reward token* that maps to a stationary reward distribution (or a scalar in the case of deterministic reward dynamics). We call U_{L+1} the *default reward token*.

Rewriting the reward dynamics in this way does not sacrifice generality since any arbitrary reward dynamics can be mapped to equation 4.1 as follows:

- Set $L = |\mathcal{S}|^2|\mathcal{A}| - 1$.
- Pick one (s, a, s') arbitrarily and map $U_{L+1} = \mathcal{R}(s, a, s')$.
- For all other L transitions map some previously unmapped $i \in [1 : L]$ to $z_j(s, a, s') = 1$ if $i = j$, otherwise $z_j(s, a, s') = 0$; and map $U_i = \mathcal{R}(s, a, s')$.

However, for many tasks we can group transitions together and therefore define the reward dynamics with $L \ll |\mathcal{S}|^2|\mathcal{A}| - 1$. This is particularly evident in tasks with sparse rewards where the agent gets a constant default reward for almost all transitions in an MDP. For example, consider the original Taxi task. In this task the agent gets a default reward of -1 for all steps except for an illegal *Pickup* action, an illegal *Dropoff* action or for reaching a terminal state. Therefore, we can represent the reward dynamics for the entire Taxi domain with $L = 3$ given the following propositions as per table 4.2:

- P_1: (s, a, s') is a transition where the *Pickup* action is applied illegally.
- P_2: (s, a, s') is a transition where the *Dropoff* action is applied illegally.
- P_3: (s, a, s') is a transition which reaches a terminal state.

Furthermore, under an object-oriented approach, each of the propositions P_1, P_2 and P_3 can be constructed from propositions over object class attributes as shown below:[7]

[7]These propositions are extracted from the state s of the transition tuple (s, a, s'). In general, the propositional statements for the reward dynamics can be extracted from s' as well. However, for this domain only (s, a) is needed to capture the reward dynamics.

Proposition	z_1	z_2	z_3	Reward Token
$P_1 = 1$	1	0	0	$U_1 = -10$
$P_2 = 1$	0	1	0	$U_2 = -10$
$P_3 = 1$	0	0	1	$U_3 = 0$
Default	0	0	0	$U_4 = -1$

Table 4.2: Expressing the reward dynamics for the Taxi domain as per equation 4.1 with $L = 3$.

- P_1: *Pickup* action with $On(\textit{Taxi}, \textit{Passenger}) = 0 \vee On(\textit{Taxi}, \textit{Passenger}) = 1 \wedge \textit{Passenger.in-taxi} = 1$.
- P_2: *Dropoff* action with $\textit{Passenger.in-taxi} = 0 \vee \textit{Passenger.in-taxi} = 1 \wedge On(\textit{Taxi}, \textit{Destination}) = 0$.
- P_3: *Dropoff* action with $On(\textit{Taxi}, \textit{Destination}) = 1 \wedge \textit{Passenger.in-taxi} = 1$.

4.5.2 Formalism

We enhance the Propositional OO-MDP formalism to include the representation of reward dynamics under equation 4.1. Given $L \in \mathbb{N}$, $a \in \mathcal{A}$ and $i \in [1 : L]$, define a set of propositions of size $D_{a,i}$:

$$\mathcal{F}_{a,i} = \{f_{a,i} : S \times S \to \mathfrak{B}\}_{i=1}^{D_{a,i}},$$

and a binary mapping function that is used determine whether reward token mapped to index i triggers given the truth values of the propositions in $\mathcal{F}_{a,i}$:

$$\mathcal{Z}_{a,i} : \mathfrak{B}^{D_{a,i}} \to \mathfrak{B}.$$

Define a set of $L + 1$ reward tokens:

$$\mathcal{U} = \{U_j\}_{j=1}^{L+1}.$$

Then the schema reward dynamics are defined by the set:

$$\mathcal{R} = \{\mathcal{F}_{a,i}, \mathcal{Z}_{a,i}, U_j | j \in [1 : L + 1], i \in [1 : L], a \in \mathcal{A}\}.$$

Then given a grounded MDP $\mathcal{M}_O = (\mathcal{S}_O, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ the reward dynamics operate as follows: the agent is currently in state $s \in \mathcal{S}_O$, takes action a and transitions

to state $s' \in \mathcal{S}_O$. For each $i \in [1 : L]$ compute the set of truth values for the propositional statements in $\mathcal{F}_{a,i}$ to get a set of binary values $\mathcal{B}_i = \{f_k(s, s')\}_{k=1}^{D_{a,i}}$. Compute $\mathcal{Z}_{a,i}(\mathcal{B}_i)$ to obtain some indicator variable z_i. The set of indicator variables is then $\mathcal{Z} = \{z_i\}_{i=1}^{L}$. The elements in the sets $\mathcal{Z}$ and $\mathcal{U}$ are then passed to equation 4.1 to compute a reward. Recall that equation 4.1 is restricted so that $\sum_{i=1}^{L} z_i \in \mathfrak{B}$. Therefore, if $\sum_{i=1}^{L} z_i = 1$ then the agent receives a reward generated from whichever reward token U_j is activated by $z_j = 1$, and if $\sum_{i=1}^{L} z_i = 0$ then the agent receives a reward generated from the default reward token U_{L+1}.

As a practical example of the formalism consider the action *Dropoff* with $\mathcal{U} = \{U_1, U_2, U_3, U_4\}$ as described in table 4.2. Suppose we have $\mathcal{F}_{Dropoff,2} = \mathcal{F}_{Dropoff,3} = \{AnyTaxiOnAnyDestination(s, s'), AnyPassengerInAnyTaxi(s, s')\}$ that are defined as: [8]

- $AnyTaxiOnAnyDestination(s, s')$: returns 1 if any objects of class *Taxi* is on the same square as any object of class *Destination* in s, and otherwise 0.
- $AnyPassengerInAnyTaxi(s, s')$: returns 1 if any objects of class *Passenger* has their *in-taxi* attribute set to 1 in s, and otherwise 0.

Given assignments $\{b_1, b_2\}$ to these propositional statements the mapping functions would be defined as:

- $\mathcal{Z}_{Dropoff,2}$: return 1 if $(b_2 = 0)$ or $(b_2 = 1$ and $b_1 = 0)$, else return 0.
- $\mathcal{Z}_{Dropoff,3}$: return 1 if $(b_1 = 1$ and $b_2 = 1)$, else return 0.

Note that we would simply define $\mathcal{F}_{Dropoff,1} = \phi$ and $\mathcal{Z}_{Dropoff,1}$ would always return a value of 0. This is because an illegal pickup is not possible when the action *Dropoff* is taken.

Suppose that the agent is currently in state s, takes action *Dropoff* and the resulting state is s'. The states s and s' are passed to $AnyTaxiOnAnyDestination(s, s')$ and $AnyPassengerInAnyTaxi(s, s')$ to obtain their respective truth values. Suppose that the resulting truth values are:

- $\{1, 0\}$: then $z_2 = 1$ and $z_3 = 0$, so the agent receives a reward from the reward token $U_2 = -10$.
- $\{1, 1\}$: then $z_2 = 0$ and $z_3 = 1$, so the agent receives a reward from the reward token $U_3 = 0$.

[8] We can actually simplify $AnyTaxiOnAnyDestination(s, s')$ and $AnyPassengerInAnyTaxi(s, s')$ to $AnyTaxiOnAnyDestination(s)$ and $AnyPassengerInAnyTaxi(s)$ respectively because these propositions do not depend on s'. However, we include s' to remain consistent with the notation of the formalism described in this section.

See Appendix Sections B.1.2 and B.2.2 for how the complete reward dynamics under this formalism are encoded for the All-Passenger Any-Destination Taxi and Sokoban domains respectively.

4.5.3 Algorithms

In this section we propose an algorithm to KWIK-learn the mapping functions $\mathcal{Z}_{a,i}$ described in Section 4.5.2 given a set of propositions $\mathcal{F}_{a,i}$ for $i \in [1 : L]$. The procedure for learning these mapping functions is analogous to learning transition dynamics as described in Section 4.4.3.

When learning transition dynamics, we use a full binary tree structure for each action and attribute with logical statements at the leaf nodes and effects at the non-leaf nodes. For learning the mapping functions, we will also use a full binary tree structure with logical statements, in the form of propositions, at the leaf nodes; but now the non-leaf nodes are binary indicators. Each such tree is used to indicate if a particular reward token is activated for a given transition. Therefore, we learn a full binary tree for each action $a \in \mathcal{A}$ and $i \in [1 : L]$.

Learning the mapping functions no longer requires the notion of an effect type, since the trees have binary leaf nodes. Therefore, in this setting, the partition function is $\Pi : \mathfrak{B} \to \mathfrak{B}$ that maps which of the binary indicator values is conjunctive and disjunctive in the tree. In a similar way to learning transition dynamics, the key to achieving efficiency under this framework is through the notion of conjunctive and disjunctive indicator values. That is, if an indicator value for action a and index i is known to occur at most at one non-leaf node in a tree then that branch can be KWIK-learned with at most $D_{a,i} + 1$ unique observations.

Returning to the example of the Taxi domain with reward dynamics as described in Section 4.5.1, $z_3 = 1$ can be efficiently learned because the precondition that maps to the activation of U_3 is the conjunction: $\mathit{On}(\mathit{Taxi}, \mathit{Destination}) = 1 \wedge \mathit{Passenger.in\text{-}taxi} = 1$. Meanwhile, $z_1 = 1$ and $z_2 = 1$ must be learned through memorisation because they occur at two leaf nodes in the tree. However, $z_1 = 0$ and $z_2 = 0$ can be efficiently learned since they occur when

$$\mathit{On}(\mathit{Taxi}, \mathit{Passenger}) = 1 \wedge \mathit{Passenger.in\text{-}taxi} = 0$$

and

$$\mathit{On}(\mathit{Taxi}, \mathit{Destination}) = 1 \wedge \mathit{Passenger.in\text{-}taxi} = 1$$

respectively.

We further assume that each U_j is KWIK-learnable with KWIK-bound B_j for $j \in [1 : L + 1]$. Under this assumption we can then KWIK-learn the complete reward dynamics with the algorithms described in this section. We introduce two algorithms that are called by the $DOORMAX_D$ algorithm presented in Section 4.6 to KWIK-learn the reward dynamics for an MDP under the Propositional OO-MDP formalism.

- *UpdateTree2* (Algorithm 9) updates the binary tree for action a and reward token mapped to index i.
- *Predict2* (Algorithm 10) makes a binary prediction for action a and reward token mapped to index i.

Note that $\hat{\mathcal{F}}(a,i)$, $\mathfrak{T}_b(a,i)$ and $\Pi(a,i)$ are initialised globally in $DOORMAX_D$. The set $\hat{\mathcal{F}}(a,i)$ contains hypothesised propositions for action a and reward token mapped to index i; the set $\mathfrak{T}_b(a,i)$ contains sets of terms that map to the binary indicator value b for action a and reward token mapped to index i; while $\Pi(a,i)$ is a partition function for action a and reward token mapped to index i that determines if an indicator value is conjunctive or disjunctive in the tree induced by $\{\mathfrak{T}_b(a,i)\}_{b\in\mathfrak{B}}$. The $\hat{}$ notation used for $\hat{\mathcal{F}}(a,i)$ indicates that these are sets of hypotheses of which the correct hypotheses must be determined by algorithm.

The Algorithm *UpdateTree2* operates in an analogous manner to the *UpdateTree1* algorithm presented in Section 4.4.3. At a high-level, the algorithm ensures that the set $\{\mathfrak{T}^g_b(a,i)\}_{b\in\mathfrak{B}}$ at all times induces a binary tree subject to the constraints of the partition function $\Pi(a,i)$. Each time it observes a set of terms $\mathcal{T}$ and an associated indicator value z it updates $\mathfrak{T}^g_z(a,i)$ to refine tree.

At the lower-level the Algorithm *UpdateTree2* updates the binary tree for action a and reward token mapped on index i given a schema state s, a resulting schema state s' and a binary indicator value z. The algorithm starts at line 1 where it computes $\mathcal{T}$, the terms for $\hat{\mathcal{F}}(a,i)$. In lines 2-3 the algorithm checks if $\mathfrak{T}_z(a,i)$ is empty, and if it is it adds $\mathcal{T}$ to $\mathfrak{T}_z(a,i)$. If $\mathfrak{T}_z$ is not empty then, as per lines 5-13, it checks if z is disjunctive or conjunctive based on the partition function $\Pi(a,i)$. If z is disjunctive then it adds $\mathcal{T}$ to $\mathfrak{T}_z(a,i)$; otherwise, if z is conjunctive then it updates the only element in $\mathfrak{T}_z(a,i)$ by removing from it any terms that mismatch terms in $\mathcal{T}$.

The Algorithm *Predict2* takes as input a reward token index i, schema state s, action a and resulting schema state s'. The algorithm then returns a binary value indicating if the reward token mapped to index i activates, or $\perp$ if the algorithm cannot yet make an accurate prediction. Once the reward dynamics are learned,

Algorithm 9: *UpdateTree2*: update binary tree for action a and reward token mapped to index i.

Input: $i \in [1:L]$, $s \in \mathcal{S}$, $a \in \mathcal{A}$, $s' \in \mathcal{S}$, $z \in \mathfrak{B}$

```
1  pass s and s' to the propositions in F̂(a,i) to retrieve a set of terms T
2  if 𝔗_z(a,i) = φ then
3  |  add T to 𝔗_z(a,i)
4  else
5  |  if Π(a,i)(z) = 0 then
6  |  |  add T to 𝔗_z(a,i)
7  |  else
8  |  |  T_temp ← T
9  |  |  T ← the only element in 𝔗_z(a,i)
10 |  |  remove from T any terms that mismatch terms in T_temp
11 |  |  𝔗_z(a,i) ← φ
12 |  |  add T to 𝔗_z(a,i)
13 |  end
14 end
```

then given a state s, action a and resulting state s', the reward token that activates can be determined by calling *Predict2* for each $i \in [1:L]$ and observing which of these returns a value of 1, or if they are all 0 then activating the default reward token. In line 1 the algorithm computes $\mathcal{T}$, the terms for $\hat{\mathcal{F}}(a,i)$. In lines 2-6 it checks whether any binary indicator value b is mapped to by $\mathcal{T}$ in $\mathfrak{T}_b(a,i)$. If it is, then b is returned; otherwise, the algorithm returns $\perp$ in line 7.

Algorithm 10: *Predict2*: prediction procedure for action a and reward token mapped to index i.

Input: $i \in [1:L]$, $s \in S$, $a \in \mathcal{A}$, $s' \in S$

```
1  pass s and s' to the propositions in F̂(a,i) to retrieve a set of terms T
2  foreach b ∈ 𝔅 do
3  |  if ∃T_b ∈ 𝔗_b(a,i) such that T_b ⊆ T then
4  |  |  return b
5  |  end
6  end
7  return ⊥
```

An important point to emphasise with regards to the *UpdateTree2* and *Predict2* algorithms is that they take the index of reward token i as one of their inputs. This information is assumed to come from the environment that the agent interacts with as per line 9 of the $DOORMAX_D$ algorithm in Section 4.6. This additional knowledge is uncharacteristic of standard RL, where the agent only receives a scalar reward at each timestep. This assumption is necessary because the algorithm needs to know which of the trees to update with an indicator value of 1 and 0 when learning reward dynamics.

This additional knowledge inherently assumes that the agent has a built-in notion

of 'reward categorisation'. To explain and motivate this, consider a human who touches a hot stove. In RL, the magnitude of the pain sensation associated with this behaviour can be used to set a scalar reward. However, humans will also segment this sensation into the 'very hot' category and learn to be cautious of touching similarly hot objects in the future.

The idea of expanding the environment reward in RL to more than just a scalar has previously been introduced and shown to have benefits both in terms of learning performance and interpretability [K.Kiguchi et al., 2003, Juozapaitis et al., 2019, van Seijen et al., 2017]. For example, by assuming that the agent receives a reward vector from its environment, where each element in the vector represents a reward type that encourages a specific behaviour (such as speed or safety), it is possible for an RL agent to learn a multi-dimensional policy that can execute any of the desired behaviours [K.Kiguchi et al., 2003]. Such an approach assumes that the agent can categorise rewards, as it is able to perceive a reward vector that comprises of different reward types from its environment.

From a practical implementation perspective, most platforms that offer off-the-shelf environments for training RL agents will only return a scalar environment reward. As such, additional processing may be required to implement RL methods that require a decomposition by reward category on such platforms.

4.6 The $DOORMAX_D$ Algorithm

In this section we combine the algorithms in Sections 4.4 and 4.5 to present the full $DOORMAX_D$ algorithm, while also providing formal guarantees of $DOORMAX_D$ efficiency for the algorithm.

4.6.1 Algorithms

This section introduces the following algorithms / procedures:

- *DefineGlobal* (Algorithm 11) defines the global data structures needed by the procedures called by $DOORMAX_D$.
- *BuildFullModels* (Algorithm 12) builds the models $\mathcal{P}_{model}$ and $\mathcal{R}_{model}$.
- $DOORMAX_D$ (Algorithm 13) is the full $DOORMAX_D$ algorithm.

Note that the inputs to $DOORMAX_D$ are assumed to be globally accessible to all the procedures called by the main algorithm.

The procedure *DefineGlobal* initialises the data structures and variables required by

DOORMAX$_D$. In lines 1-8 the procedure initialises the $\hat{\mathcal{F}}(a, C.\alpha)$, $\hat{\mathcal{G}}(a, C.\alpha)$ and $\mathfrak{T}_e^g(a, C.\alpha)$ data structures as well as the variable M required for learning transition dynamics. In lines 9-14 it initialises the $\hat{\mathcal{F}}(a, i)$ and $\mathfrak{T}_b(a, i)$ data structures as well as the partition functions $\Pi(a, i)$ and reward tokens $\hat{U}_j$ for learning reward dynamics.

Algorithm 11: *DefineGlobal*: define global variables and data structures required for *DOORMAX*$_D$.

```
1  foreach C ∈ 𝔈 do
2      foreach (a, C.α) ∈ 𝒜 × Att(C) do
3          define F̂(a, C.α) globally as the hypothesised deictic predicates for each
             action a and attribute C.α
4          define Ĝ(a, C.α) globally as the hypothesised effect types for each action a
             and attribute C.α a
5          foreach g ∈ Ĝ(a, C_α) do
6              foreach e ∈ g.ℰ do
7                  define 𝔗_e^g(a, C.α) globally as the set of sets for each action a, attribute
                     C.α and effect e
8  define M > 1 such that M − 1 is the maximum number of elements allowed in any
     set 𝔗_e^g(a, C.α)
9  foreach (a, i) ∈ 𝒜 × [1 : L] do
10     define F̂(a, i) globally as the hypothesised propositions for each action a and
         index i mapped to z_i
11     define 𝔗_b(a, i) globally a set of sets for each action a, index i mapped to z_i and
         Boolean b
12     define Π(a, i) globally as a partition function for each action a and index i
         mapped to z_i
13 foreach j ∈ [1 : L + 1] do
14     define Û_j globally as the estimated reward token mapped to index j
```

The Algorithm *BuildFullModels* builds the full models $\mathcal{P}_{model}$ and $\mathcal{R}_{model}$. The algorithm starts at lines 1-2 where it initialises the models to the most optimistic case where every transition returns r_{max}. In line 3 the algorithm iterates over all tuples $(s, a) \in \mathcal{S}_O \times \mathcal{A}$ of the grounded MDP. In lines 4-15 the algorithms calls *Predict1* for every object attribute value $o.\alpha$ in s and checks whether it can accurately predict the object attribute value $o.\alpha'$ when taking action a in order to construct the next state s'. If it can, then the algorithm proceeds to lines 16-25 where it calls *Predict2* on the tuple (s, a, s') for every reward token index $i \in [1 : L]$. The algorithm checks if it can make an accurate prediction for every $i \in [1 : L]$. If it can, the algorithm proceeds to lines 26-31 where it selects the activated reward token U_j. If U_j has been KWIK-learned, then the algorithm sets $\mathcal{P}_{model}(s, a)$ to s' and $\mathcal{R}_{model}(s, a, s')$ to $\hat{U}_j$.

The *DOORMAX*$_D$ algorithm is the root algorithm of this chapter. Being a KWIK-R_{max} based algorithm, it follows the template of Algorithm 6 presented in Section

Algorithm 12: *BuildFullModels*: build $\mathcal{P}_{model}$ and $\mathcal{R}_{model}$ for $DOORMAX_D$.

initialise $\mathcal{P}_{model}(s,a) \leftarrow \perp$ for all $(s,a) \in \mathcal{S}_O \times \mathcal{A}$
initialise $\mathcal{R}_{model} \leftarrow r_{max}$ for all $(s,a,s') \in \mathcal{S}_O \times \mathcal{A} \times \mathcal{S}_O$
foreach $(s,a) \in \mathcal{S}_O \times \mathcal{A}$ **do**
    $s' \leftarrow s$
    allPredKnown1 $\leftarrow 1$
    foreach *object* o *in* s' **do**
        foreach *attribute* $C.\alpha \in Att(C)$ *from the class* C *of object* o **do**
            pred $\leftarrow$ *Predict1*$(C.\alpha, s, o, a)$
            if *pred* $= \perp$ **then**
                allPredKnown1 $\leftarrow 0$
            else
                replace attribute $C.\alpha$ of object o in s' with *pred*
            end
        end
    end
    if *allPredKnown1* $= 1$ **then**
        allPredKnown2 $\leftarrow 1$
        $j \leftarrow L + 1$
        foreach $i \in [1 : L]$ **do**
            pred $\leftarrow$ *Predict2*(i, s, a, s')
            if *pred* $= \perp$ **then**
                allPredKnown2 $\leftarrow 0$
            else if *pred* $= 1$ **then**
                $j \leftarrow i$
        end
        if *allPredKnown2* $= 1$ **then**
            if $\hat{U}_j$ *with KWIK-bound* B_j *has been KWIK-learned* **then**
                $\mathcal{P}_{model}(s,a) \leftarrow s'$
                $\mathcal{R}_{model}(s,a,s') \leftarrow \hat{U}_j$
            end
        end
    end
end
return ($\mathcal{P}_{model}$, $\mathcal{R}_{model}$)

4.2.5. In line 1 it calls *DefineGlobal* to initialise the required data structures and variables. A start state s of the MDP is initialised in line 3 and the agent-environment interaction starts at line 4 with a while loop that ends when a terminal state is reached. In line 5 the models $\mathcal{P}_{model}$ and $\mathcal{R}_{model}$ are built by calling *BuildFullModels*. In line 6 the algorithm constructs an MDP $\hat{M}$ with transition dynamics $\mathcal{P}_{model}$ and reward dynamics $\mathcal{R}_{model}$. In line 7 the algorithm computes an optimal policy $\hat{\pi}_*$ for $\hat{M}$ by running an exact planning algorithm. In lines 8-9 the algorithm selects an action from $\hat{\pi}_*$ to observe a next state s', reward r and reward token index j from the environment. In lines 10-15 the algorithm calls *UpdateTree1* to update the appropriate binary trees for every $o.\alpha'$ in s'. In lines 16-22 the algorithm calls *UpdateTree2* to update the appropriate binary trees for every reward token index $i \in [1 : L]$. In lines 23-25 the algorithm updates $\hat{U}_j$ with r if $\hat{U}_j$ is not yet KWIK-learned. The loop ends after line 26 by setting the current state s to the next state s' for the start of the next iteration.

Algorithm 13: $DOORMAX_D$: the $DOORMAX_D$ algorithm for episodic tasks.

Input: $\mathfrak{C}$, L, $\mathcal{S}_O$, $\mathcal{A}$, γ, r_{max}

```
1  DefineGlobal()
2  repeat
3      start episode at initial state s
4      while s ∉ S_terminal do
5          (P_model, R_model) ← BuildFullModels()
6          define an MDP M̂ = (S_O, A, P_model, R_model, γ)
7          compute an optimal policy π̂_* for M̂ using exact planning Algorithm
8          choose action a ∈ A(s) from π̂_*
9          observe some next state s' and reward r generated from reward token j
10         foreach object o in s do
11             foreach attribute C.α ∈ Att(C) from the class C of object o do
12                 o.α' ← value of attribute C.α for object o in s'
13                 UpdateTree1(C.α, s, o, a, o.α')
14             end
15         end
16         foreach i ∈ [1 : L] do
17             z ← 0
18             if i = j then
19                 z ← 1
20             end
21             UpdateTree2(i, s, a, s', z)
22         end
23         if Û_j with KWIK-bound B_j has not been KWIK-learned then
24             update Û_j with r
25         end
26         s ← s'
27     end
28 until forever
```

4.6.2 Theory

In this section we present a proof that the learning algorithm presented in Section 4.4.3 KIWK-learns the transition dynamics for action a and attribute $C.\alpha$, as well as derive the corresponding KWIK-bound. The proof is analogous to that of Propositional OO-MDPs [Diuk et al., 2008, Diuk, 2010].

Theorem 2 (KWIK-bound under $DOORMAX_D$)**.** *For action a and attribute $C.\alpha$ let $\hat{F}$ be a set of size D that contains hypothesised deictic predicate preconditions on which the transition dynamics of $C.\alpha$ when taking action a may depend. Let $\hat{\mathcal{G}} = \{g_i\}_{i=1}^{N}$ be a set of size N that contains hypothesised effect types where each $e \in g_i.E$ has domain $Dom(C.\alpha)$. Let $K_0 = \max_{g \in \hat{\mathcal{G}}} K_0^g$ and $K_1 = \max_{g \in \hat{\mathcal{G}}} K_1^g$. Let $\mathcal{H} = \{Tree(g, \hat{F}, M) | g \in \hat{\mathcal{G}}\}$ for some constant $M > 1$. Then if exactly one $h^* \in \mathcal{H}$ is true, the transition dynamics for a and $C.\alpha$ can be KWIK-learned under the $DOORMAX_D$ algorithm with KWIK-bound $N(K_0 M + K_1(D+1)+1) + N - 1$.*

Proof. Consider an effect type $g \in \hat{\mathcal{G}}$. If this is the correct effect type then it can be learned with KWIK bounds $K_0^g M + K_1^g(D+1)$. This is because the K_1^g conjunctive effects require at most $D+1$ observations each to learn the terms they depend on while the terms for the K_0^g disjunctive effects can be memorised M times each. If we then consider all N effect types in $\hat{\mathcal{G}}$, an upper bound on the number of observations required so that all effect types are either removed or return some prediction is $N(K_0 M + K_1(D+1)+1)$.

Now when we call *Predict1* if two effect types provide a prediction that is not the same we remove one of them on the subsequent run of the *UpdateTree1* procedure and this can occur at most $N-1$ times. This gives a total KWIK-bound of $N(K_0 M + K_1(D+1)+1) + N - 1$.

□

An analogous proof can be constructed for the learning algorithm of reward dynamics presented in Section 4.5.3 showing that if a binary value occurs at most once in the tree for action a and index i mapped to z_i, then that branch can be learned with KWIK-bound $D_{a,i} + 1$.

4.7 Experiments

In this section we conduct experiments to illustrate the benefits of the object-oriented frameworks presented in this chapter. In Section 4.7.1 we conduct experiments on the All-Passenger Any-Destination Taxi domain to efficiently learn transition dynamics, under the assumption of known reward dynamics. In Section

4.7.2 we conduct experiments on the All-Passenger Any-Destination Taxi domain to efficiently learn reward dynamics, under the assumption of known transition dynamics. In Section 4.7.3 we demonstrate that the reward dynamics and transition dynamics can be efficiently learned together for the Sokoban domain. In all experiments we show that zero-shot transfer is possible in these domains, thus leading to improved sample efficiency under our proposed frameworks.

4.7.1 All-Passenger Any-Destination Taxi Domain: Learning Transition Dynamics

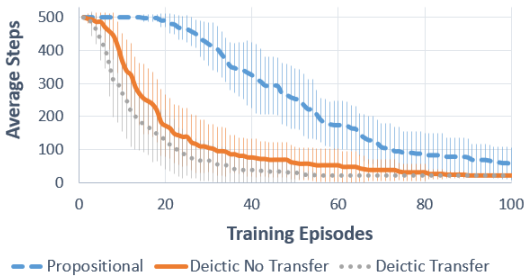
We conduct two sets of experiments on the All-Passenger Any-Destination Taxi domain under the assumption that reward dynamics are known while transition dynamics need to be learned. In the first set of experiments we have one destination and we fix the number of passengers, n. We generate a grounded MDP with an initial state by randomly sampling n passenger locations and one destination location from one of six pre-specified locations and we also sample a random taxi start location together with one of four wall configurations as shown in Figure 4.4a.

We apply 20 independent runs of the following procedure: we sample 10 test MDPs with random initial states. We then randomly sample a training MDP and run $DOORMAX_D$ on it for one episode until we reach a terminal state. Upon termination, we test performance by running $DOORMAX_D$ for one episode on each of the 10 test MDPs, stopping an episode early if we exceed 500 steps. We repeat this for 100 training MDPs. Since all the MDPs come from the same schema we can share transition dynamics between our MDPs - but we only update the transition dynamics on training MDPs.

In our experiments we start with $n = 1$ passenger and incrementally increase to $n = 4$ passengers. We run our experiments for Propositional OO-MDPs and two versions of Deictic OO-MDPs. In the first, without transfer, we relearn the transition dynamics for each n while for the second, with transfer, we transfer the previously learned transition dynamics each time we increase n. We report results in Figure 4.8 that averages over the 20 independent runs the average number of steps for the 10 test MDPs with standard deviation error bars included.

We see from the results that for this domain, Deictic OO-MDPs outperform Propositional OO-MDPs as we increase the number of passengers. This is because with Propositional OO-MDPs we need to add more propositions to the preconditions as we increase the number of passengers - in fact the propositional representation is unable to learn the task when $n = 4$ even after 100 training episodes. Furthermore, as the MDPs belong to the same schema it is beneficial to transfer the previous tran-

(a) One passenger experimental results.



(b) Two passenger experimental results.

(c) Three passenger experimental results.

(d) Four passenger experimental results.

Figure 4.8: Experimental results for learning transitions dynamics in the All-Passenger Any-Destination Taxi domain with different number of passengers.

sition dynamics under the deictic representation. Specifically, once we get to $n = 4$ passengers the transition dynamics we transfer from the $n = 3$ experiment have learned the schema transition dynamics completely and we have zero-shot transfer of the transition dynamics.

We observe from Figure 4.8 that using the deictic representation without transfer is actually learning slightly faster as we add more passengers. This is somewhat misleading. What is actually happening is that as the tasks become more complex, the agent is able to learn more about the transition dynamics over a single training episode, but that episode will require many more steps to complete. To illustrate this, and also highlight the robustness of the deictic representation with transfer methodology, we conduct a second set of experiments. These experiments are similar to those conducted for the first set but we now use a larger 10×10 gridworld with five passengers and three destinations as in Figure 4.4b. In these experiments we stop after 100 steps for each episode of the training MDPs. Furthermore, for the deictic representation with transfer we simply transfer the learned transition dynamics of $n = 4$ passengers and do no additional learning on the new larger gridworld.

In Figure 4.9 we plot for all the experiments run the average number of steps relative to optimal number of steps. We see that the deictic representation with transfer is able to solve the larger gridworld optimally with no additional learning of the transition dynamics. Meanwhile the deictic representation with no transfer, which

was decreasing up to $n = 4$, now has a jump between at $n = 5$ because the agent does not have the benefit of learning for a full training episode. We cap the graph's y axis at 10 to make it more readable, but remark that Propositional OO-MDPs exhibits exponentially worse performance relative to optimal as the tasks become more complex.

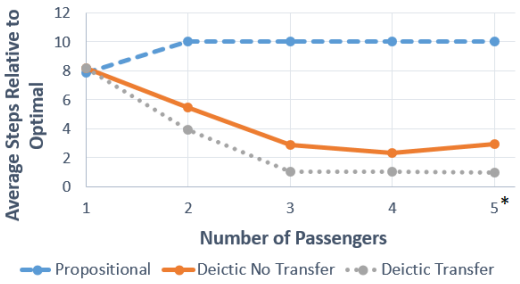


Figure 4.9: Average number of steps relative to optimal number of steps as we add more passengers - for $n = 5$ we also increase the gridworld size and add more destinations hence it is marked with a $*$.

We provide more details on these experiments as well as a full list of the hypothesised deictic predicates used for each action and attribute in Appendix Section B.1.3.

4.7.2 All-Passenger Any-Destination Taxi Domain: Learning Reward Dynamics

In this section, we run a similar experiment to that of Section 4.7.1, but we now assume known transition dynamics and focus on learning and transfer of reward dynamics.

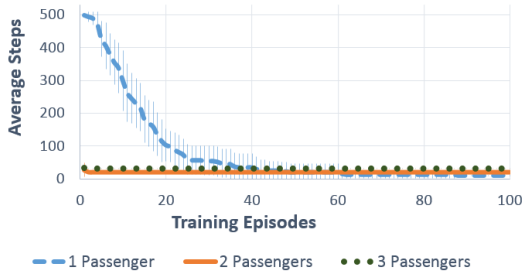


Figure 4.10: Experimental results for learning reward dynamics in the All-Passenger Any-Destination Taxi domain with different number of passengers.

In Figure 4.10 we see that, as in the case of learning transition dynamics, we can transfer the model of the reward dynamics between tasks of the domain as more passengers are added. The reward dynamics are completely learned after completing the $n = 2$ passenger MDPs.[9] From there, the reward dynamics are zero-shot

[9]While it appears from the figure that the reward dynamics are completely learned after $n = 1$,

transferred to the $n = 3$ passenger MDPs. In fact, the model of the reward dynamics can be zero-shot transferred to any MDP of the All-Passenger Any-Destination Taxi domain schema after learning on the $n = 2$ passenger MDPs.

We provide more details on these experiments as well as a full list of the hypothesised propositions used for each action and reward token in Appendix Section B.1.4.

4.7.3 Sokoban Domain

To demonstrate the benefits of Deictic OO-MDPs, we conduct an experiment on a more challenging Sokoban domain. In this experiment we first learn both the transition dynamics and reward dynamics by randomly sampling MDPs with start states as shown in Figure 4.11a and continue learning on these until we have no $\perp$ predictions in these MDPs. These simple MDPs, each with approximately 8000 states, are enough to completely learn the schema transition and reward dynamics of this domain under the object-oriented frameworks presented in this chapter.

Once learned we zero-shot transfer the transition dynamics to a more complex Sokoban task as shown in 4.11b. This task comes from the 'Micro-Cosmos' level pack and has approximately 10^6 states while the optimal number of steps to solve this task 209. With no additional learning we run value iteration and solve for an optimal policy. Note that the ability to transfer here is critical. The larger MDP has approximately 125 times more states than the toy MDPs. Running R_{max} based algorithms directly on the larger MDP is very slow because at each step it is required to compute a policy with an exact planning algorithm and this has high computational complexity. By transferring the transition dynamics learned in the toy MDPs we can solve the larger MDP with only a *single run* of value iteration which is highly sample efficient.

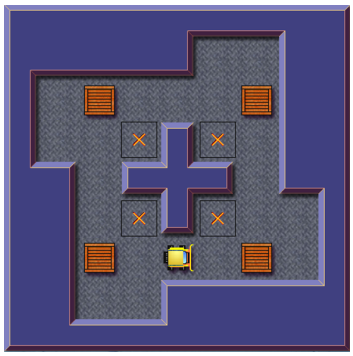
We include additional details on the transition dynamics and reward dynamics for this domain in Appendix Section B.2.

4.8 Future Work

The work presented in this chapter has shown that object-oriented frameworks offer a promising direction to overcome many of the major challenges in RL. The algorithms presented in this chapter learn a compact representation of the transition

there is a small amount of learning required for $n = 2$ as the agent needs to learn the dynamics when picking up a passenger while another passenger is already in the taxi. It is not clearly visible on the figure because this learning happens in the first few training episodes and does not have a material impact on the optimal number of steps to solve the test tasks.

(a) Four simple Sokoban tasks.



(b) A more complex Sokoban task with four boxes from the 'Micro-Cosmos' level pack.

Figure 4.11: Training tasks and test task for Sokoban.

and reward dynamics from a small number of simple source tasks of a domain, that are transferable across that domain.

While promising, a major shortcoming with the work presented in this chapter is the amount of prior domain knowledge required to define the formalisms. For example, the Deictic OO-MDP formalism described in Section 4.4.1 requires for each action and object class attribute a set of deictic predicates and effects to define the domain's transition dynamics. Taking a further step back, we have assumed prior knowledge of the objects classes and object class attributes that make up the domain.

In order to implement the frameworks of this chapter, all these components need to be carefully specified; and this is a time-consuming, error-prone process. While some research in object-oriented representations has focused on learning the lower-level building blocks required for object-oriented representations to function [Hershkowitz et al., 2015, Yitao et al., 2016], most research assumes this knowledge as given. Therefore, an important direction for future research is the development of algorithms that learn the lower-level building blocks of object-oriented representations through experience.

Sections 4.4.3, 4.5.3 and 4.6 outline $DOORMAX_D$, a provably efficient learning algorithm for the object-oriented frameworks described in this chapter. The algorithm makes use of binary tree structures to store the dynamics of a domain. However, the requirements for efficiency in $DOORMAX_D$ are restrictive, relying on unique conjunctive effects to drive efficient learning. In practice, domains may have few, if any, conjunctive effects making the algorithm unsuitable. Coupled with this, the algorithm lacks robustness - conjunctive effects are required to be specified upfront, as are the logical statements that make up the preconditions. If any of these are incorrectly specified, the algorithm fails. Therefore, an important area of future research is the development of provably efficient learning algorithms that are less

restrictive and have built-in fall-back mechanisms.

A further interesting direction for future research is the connection between $DOORMAX_D$ (that relies on binary tree structures) and traditional decision trees used in supervised learning for regression and classification tasks. Previous research has utilised such decision trees in factored MDPs to show that they have similar benefits to $DOORMAX_D$ in terms of sample efficiency and transferability, while also having an advantage of requiring far fewer assumptions in order to be applied [Hester and Stone, 2009]. However, this sample efficiency gain is demonstrated only empirically, whereas $DOORMAX_D$ comes with theoretical KWIK-bound guarantees. An in-depth analysis comparing these two approaches may provide important insights and produce techniques that combine the strength of both approaches.

The $DOORMAX_D$ algorithm presented in this chapter applies only to environments with deterministic transition dynamics. In the most general RL setting, the transition dynamics are stochastic. The original $DOORMAX$ algorithm was first introduced for environments with deterministic transition dynamics, but was later extended to a family of stochastic transition dynamics [Diuk, 2010]. Extending $DOORMAX_D$ to this family, and possibly other families, is an area of future work.

Finally, in our experimental section, we show that by learning on some simple source tasks of a domain we can learn the complete dynamics for the entire domain. The set of source tasks chosen in the All-Passenger Any-Destination and Sokoban experiments were carefully constructed to ensure they cover the complete domain dynamics. For example, Sokoban requires tasks with at least two boxes and multiple walls at specific locations in order to ensure that the interactions between the warehouse keeper, boxes and walls can be fully learned. However ideally, we do not want to construct these source tasks manually. Rather we would want to utilise an algorithm that generates optimal source tasks to learn from so that the next generated task is the simplest possible task to solve, from which the most possible knowledge can be acquired. The idea constructing a set of source tasks to learn from in such an incremental manner forms part of the broader research field of Curriculum Learning in RL [Narvekar et al., 2020].

4.9 Conclusion

This chapter has introduced the Deictic Object-Oriented framework for RL, a novel object-oriented framework for representing and efficiently learning the transition dynamics for certain classes of domains through the notion of a lifted schema. The representation is based on the concept of a deictic predicate, whereby a central grounded reference object relates itself to lifted object classes. Since only the refer-

ence object is grounded, the transition dynamics under this formalism can transfer across all grounded MDPs from the schema.

The deictic object-oriented formalism introduced in this chapter is more general than previously introduced propositional object-oriented formalisms, and is able to compactly represent the transition dynamics of domains not possible under a propositional approach. Such domains include the All-Passenger Any-Destination Taxi domain introduced in this chapter, as well as the well-known Sokoban domain.

Furthermore, previous work on Propositional OO-MDPs assumed that transition dynamics need to be learned, while reward dynamics are known to the agent. In this chapter we have also introduced a propositional object-oriented framework to represent and efficiently learn the reward dynamics for certain classes of domains.

We then combine these individual learning algorithms to introduce a KWIK-R_{max} based algorithm called $DOORMAX_D$ and formally derive its KWIK-bound. In totality we show that, for certain classes of domains, it is possible to both compactly represent and KWIK-learn both the transition and reward dynamics for the entire domain represented through a schema.

The advantage of this approach is that we can learn these dynamics on some source MDPs from the schema and then transfer them to a new target MDP from the schema. This transfer procedure improves sample efficiency because the agent can reuse the models it has previously learned from the source MDPs, and so does not need to waste samples relearning this knowledge for each new MDP. In fact, once the full models have been learned, we can zero-shot transfer them across MDPs of the schema.

In terms of the complexity metrics discussed in Section 2.3.1, the algorithms introduced in this chapter reduce learning complexity in two senses: firstly, because the algorithms leverage the KWIK framework they have provably efficient KWIK-bounds. Secondly, because the learned models transfer between tasks of the same domain, once the models are learned from some source tasks of the domain there is no learning required to at all. Unfortunately, the KWIK-R_{max} requires an exact planning algorithm to run at every step, and such algorithms tend to have high computational complexity.

To illustrate the advantages of our approach we run experiments on the All-Passenger Any-Destination Taxi domain as well as the Sokoban domain. In these experiments we show that it is possible to efficiently learn the full models on a set of simple source MDPs and then zero-shot transfer them to more complex MDPs for sample efficient learning.

Chapter 5

Efficient Learning of Likely-Admissible Heuristics

The work presented in this chapter also appears in: Marom and Rosman [Marom and Rosman, 2020].

5.1 Introduction

In Chapter 4 we introduce $DOORMAX_D$, an efficient model-based Reinforcement Learning (RL) algorithm that takes advantage of object-oriented representations to learn an optimal policy for a Markov Decision Process (MDP). The $DOORMAX_D$ algorithm is efficient in terms of learning complexity because it learns the models of the transition and reward dynamics for an entire domain from some set of small source tasks of that domain.

As shown in the experimental section of Chapter 4, once the models are learned they can be zero-shot transferred to a large target task of the domain. At this point, computing an optimal policy for the target task can be treated as a planning problem. This is because the agent does not need to further interact with the environment to learn the dynamics models. The bottleneck in finding an optimal policy for the target task then boils down to efficient planning. Unfortunately, as discussed in Section 2.2.2, naïve planning algorithms can have high computational complexity in large tasks because of the 'curse of dimensionality'.

In the RL literature, planning tasks have historically relied on Dynamic Programming (DP) algorithms such as value iteration [Sutton and Barto, 2018]. However, planning actually has its roots in graph traversal algorithms [Russell and Norvig, 1995]. Many graph traversal algorithms are heuristic-based [Pearl, 1984]. A heuristic

function incorporates prior knowledge into a planning task by producing a quick-to-compute estimate of the optimal cost-to-goal from any state. Provided the heuristic is well-specified, it can be used to guide search by prioritising operators that lead to states with low cost-to-goal estimates. Such a heuristic-based approach to planning has been shown to greatly reduce computational complexity [Korf and Taylor, 1996, Korf, 1997].

Furthermore, certain heuristic-based graph traversal planning algorithms, such as A* [Hart et al., 1968] and IDA* [Korf, 1985], have guarantees of optimality provided that the heuristic is admissible - that is, the heuristic never overestimates the true cost-to-goal. Given a strong admissible heuristic, such algorithms have been shown to find optimal plans for domains with enormous state-spaces [Korf and Taylor, 1996, Korf, 1997].

The A* and IDA* algorithms operate in the *classical planning* setting. This setting is a simpler setting than the general finite MDP setting, as further discussed is Section 5.2.1. One of the major simplifications of the classical planning setting is the assumption of deterministic transition dynamics. More recently, ideas from DP and heuristic-based search have been combined to solve tasks with stochastic transition dynamics making heuristic-based search applicable to a broader class of MDPs that what is allowed under classical planning [Hansen and Zilberstein, 1998, Hansen and Zilberstein, 2001, Dai et al., 2011].

While heuristic-based search algorithms can lower computational complexity, their performance depends largely on the quality of the heuristic provided. Unfortunately, crafting strong admissible heuristics is often difficult, requiring expert domain knowledge and high memory resources [Junghanns and Schaeffer, 2001, Felner and Adler, 2005, Zahavi et al., 2008, Slaney and Thiébaux, 2001, Haslum and Scholz, 2003, Helmert, 2010].

An alternative approach to crafting heuristics by hand is to learn them from data using machine learning algorithms [Arfaee et al., 2010, Arfaee et al., 2011, Ernandes and Gori, 2004, Groshev et al., 2017, Thayer et al., 2011, Samadi et al., 2008, McAleer et al., 2018]. Previous research has shown that it is possible to learn a heuristic from a set of optimal plans of tasks from some domain and then transfer the learned heuristic to a new task of the same domain [Ernandes and Gori, 2004]. Such heuristics have shown to perform well. However, this approach does not guarantee that the heuristic is admissible, and so non-optimal plans can be produced by planners that use such heuristics.

Furthermore, for domains with large state-spaces, it is often prohibitive to produce datasets of optimal plans for learning. An alternative approach is to apply a

bootstrap learning procedure that interweaves planning and learning [Arfaee et al., 2010, Arfaee et al., 2011]. Under a bootstrap learning approach, tasks of varying difficulty from a domain are generated online. The current version of the heuristic is used to try solve each of these tasks given some constraint (such as a time limit). The heuristic is then updated by learning from the plans produced by the solved tasks, thus becoming a stronger heuristic.

The advantage of the bootstrap approach is that there is no need to start with an offline dataset of plans. This approach has been shown be effective in domains with large state-spaces, where generating plans to learn from is prohibitive [Arfaee et al., 2010, Arfaee et al., 2011]. However, previous approaches to bootstrap learning have two major shortcomings: 1) they generate tasks randomly which is inefficient and; 2) the final heuristic they output produces high suboptimality because errors compound with each iteration of the procedure.

5.1.1 Contributions

In this chapter we propose to overcome the limitations of previous bootstrap learning approaches by leveraging measures of *epistemic* and *aleatoric* uncertainty. Epistemic uncertainty accounts for uncertainty in the model of a process and can be explained away given enough data, while aleatoric uncertainty accounts for uncertainty that the model cannot explain away due to probabilistic variation in the data.

Epistemic uncertainty can be used to efficiently explore task-space so as to generate training tasks that are of the right level of complexity. This approach is more sample efficient than random task generation because tasks are generated in accordance with how much new knowledge the current heuristic can gain from solving the task and subsequently learning from the task's plan.

Meanwhile, epistemic and aleatoric uncertainty are combined in our proposed method so that the heuristic remains *likely-admissible* during planning. Likely-admissible heuristics have previously been introduced as heuristics that are optimal with some probability, and this leads to likely-optimal plans [Ernandes and Gori, 2004]. Using such a heuristic mitigates the compounding error problem of previous bootstrap approaches because the heuristic learns from plans that are optimal with high probability at each iteration. This then produces a final heuristic that produces low suboptimality.

In this chapter we propose a conceptual framework that allows for a broad range of models to be used for the purpose of efficiently learning likely-admissible heuristic. We then introduce a practical implementation based on Bayesian neural networks for experimentation. We run experiments on the 15-puzzle, 24-puzzle, 24-pancake and

15-blocksworld domains using this implementation and illustrate empirically that our approach is more sample efficient than random task generation, while outputting a final heuristic that produces plans with low suboptimality.

We emphasise that the focus on this chapter is on learning heuristics, while remaining agnostic to the choice of heuristic-based planner used. We therefore restrict our attention to the simpler setting of classical planning to illustrate the validity of the learning approach and we run experiments with IDA* on classical planning domains. However, our approach is expected to work when using modern heuristic-based planning algorithms like LOA* that operate in the more general MDP setting [Hansen and Zilberstein, 2001].

5.1.2 Chapter Structure

This chapter is organised as follows: in Section 5.2 we introduce the relevant background for heuristic-based planning, learning heuristics from data and modelling uncertainty with Bayesian methods; in Section 5.3 we introduce a conceptual framework to efficiently learn likely-admissible heuristics that can work with any model that produces measures of epistemic and aleatoric uncertainty; in Section 5.4 we introduce a practical implementation of the conceptual framework based on Bayesian neural networks; in Section 5.5 we conduct experiments on the 15-puzzle, 24-puzzle, 25-pancake and 15-blocksworld domains; and we end the chapter with a discussion of future work in Section 5.6 and concluding remarks in Section 5.7.

5.2 Background

5.2.1 Terminology and Conventions in Classical Planning

The fields of RL and planning are strongly connected, and often crossover. However, these two fields have distinct histories with separate terminologies and conventions. As this chapter focuses on classical planning, we switch to the classical planning terminology and conventions as further described in this section:

- We refer to a plan rather than a policy.
- We refer to a goal state rather than a terminal state.

In RL, as per Section 2.1.1, an RL task is typically defined by a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ with a start state distribution ρ that describes an MDP. In classical planning, a planning task is defined by a tuple $\mathcal{T} = (\mathcal{S}, \Omega, \mathcal{E}, \mathcal{C}, s_0, \mathcal{S}_g)$ [Russell and Norvig, 1995], where:

- $\mathcal{S}$ is a finite set of states;

- Ω is a finite set of operators, while $\Omega(s)$ returns the legal operators than can be executed in state $s \in \mathcal{S}$;
- $\mathcal{E}(s, o)$ is an effect function that returns the next state $s' \in \mathcal{S}$ when operator $o \in \Omega(s)$ is applied in s;
- $\mathcal{C}(s, s')$ is a strictly positive and bounded real-valued function for the cost of moving from state s to s';
- $s_0 \in \mathcal{S}_0 \subseteq \mathcal{S}$ is a start state from the set of possible start states $\mathcal{S}_0$; and
- $\mathcal{S}_g \subseteq \mathcal{S}$ is a set of possible goal states.

A planning task $\mathcal{T}$ and an MDP $\mathcal{M}$ are related as follows:

- The state-space $\mathcal{T}.\mathcal{S}$ and operators $\mathcal{T}.\Omega$ are identical to the state-space $\mathcal{M}.\mathcal{S}$ and action-space $\mathcal{M}.\mathcal{A}$.
- The effect function $\mathcal{T}.\mathcal{E}$ plays a similar role to the transition dynamics $\mathcal{T}.\mathcal{P}$ except that $\mathcal{T}.\mathcal{E}$ is deterministic while $\mathcal{T}.\mathcal{P}$ can be stochastic.
- The cost function $\mathcal{T}.\mathcal{C}$ plays a similar role the reward dynamics $\mathcal{M}.\mathcal{R}$ except that 1) the objective in planning is to minimise cost, while the objective in RL is to maximise reward; 2) $\mathcal{T}.\mathcal{C}$ is a strictly positive and bounded function that maps to a scalar cost, whereas $\mathcal{M}.\mathcal{R}$ can map to any stationary distributions with finite variance; and 3) $\mathcal{T}.\mathcal{C}$ is a function of s and s' only whereas $\mathcal{M}.\mathcal{R}$ can depend on an action as well.
- The set of start states $\mathcal{T}.\mathcal{S}_0$ play as similar role to
- The set of goal states $\mathcal{T}.\mathcal{S}_g$ play a similar role to the set of terminal states of $\mathcal{M}$ (assuming $\mathcal{M}$ is episodic).
- $\mathcal{M}$ can discount future rewards with $\mathcal{M}.\gamma$ whereas $\mathcal{T}$ has no notion of discounting.

Based on the above, we see that the classical planning setting is more restrictivem than the MDP setting.

A planning domain is then defined by the tuple $\mathfrak{D} = (\mathcal{S}, \Omega, \mathcal{E}, \mathcal{C}, \mathcal{S}_0, \mathcal{S}_g)$. Given a planning domain, a planning tasks can be instantiated by selecting a particular start state $s_0 \in \mathcal{S}_0$.

In classical planning, the effect function is deterministic. Therefore, a plan π can be described as a sequence of states starting from s_0 and ending in $s_g \in \mathcal{S}_g$ i.e. $\pi = (s_0, s_1, s_2, ..., s_n = s_g)$. The objective in planning is to find an optimal plan π_* that incurs minimal cost.

5.2.2 Heuristic-Based Planning

The value iteration algorithm described in Section 2.1.3 is one of the most well-known algorithms for planning in RL. Value iteration is a simple DP algorithm that works by sweeping over the state-space of an MDP and applying Bellman updates as per equation 2.2 until convergence. Unfortunately, value iteration has high computational complexity which is worst-case $\mathcal{O}(|A||S|^2)$ per sweep. This makes value iteration an infeasible planning algorithm for any practical task with large action-space or state-space.

A general way to overcome this high computational complexity is to augment the planner with prior knowledge in the form of a heuristic function $h(s)$ [Pearl, 1984]. Such a heuristic function returns an estimate of the optimal cost-to-goal from any state $s \in \mathcal{S}$. Such a heuristic can be used to guide search to relevant parts of the state-space by prioritising operators that lead to states with low cost-to-goal estimates and this can greatly improve search performance.[1]

Denote by $h^*(s)$ the optimal, and unknown, cost-to-goal from s. An important property that underlies many heuristic-based planning algorithms is that if h is an *admissible heuristic* so that $h(s) \leq h^*(s)$ for all $s \in \mathcal{S}$ then the planner is guaranteed to return a plan π_* that is optimal in the sense that it incurs minimal cost. A* and its variant IDA* are examples of such algorithms.[2]

A trivial admissible heuristic for any planning domain is the zero-heuristic, $h(s) = 0$ for all $s \in \mathcal{S}$. While admissible, the zero-heuristic is uninformative and therefore does not aid in reducing computational complexity. However, using a stronger admissible heuristic can greatly reduce computational complexity. A natural question that arises with admissible heuristics is what constitutes a 'strong' admissible heuristic in the first place. A seemingly intuitive definition is that the closer an admissible heuristic h is to h^*, the stronger h is. However, this turns out not to be the case, as it is possible to define weaker heuristics that perform better than stronger heuristics under this definition [Holte, 2010]. Nevertheless, there do exist theoretical guarantees that stronger heuristics under his definition reduce the upper bound on the number of nodes expanded by A* [Dinh et al., 2007]. Coupled with the fact that if $h = h^*$ then A* expands exactly $d_* = |\pi_*|$ nodes, a common target when constructing admissible heuristics is to construct h to be as close to h^* as

[1] For ease of notation we restrict the exposition of this chapter to domains with unique goal states, and so h is a function of s only. We discuss how to extend to multiple goal states in Section 5.6.

[2] A major disadvantage of A* is its high space complexity which is worst-case $\mathcal{O}(|\Omega|^{d_*})$, where $d_* = |\pi_*|$ is the optimal solution depth. The IDA* algorithm is a variant of A* that has a worst-case space complexity of $\mathcal{O}(d_*)$ making it a good choice for domains with large state-spaces that tend to have deep optimal solutions.

possible.

While admissibility is a desirable property, guaranteeing admissibility of a heuristic is often difficult. An alternative property is to require that the heuristic be likely-admissible [Ernandes and Gori, 2004]. A likely-admissible heuristic, denoted h^α, is admissible with probability $\alpha \in (0, 1)$ i.e. $P(h^\alpha(s) \leq h^*(s)) = \alpha$ for all $s \in \mathcal{S}$. Since only overestimations of states within the optimal plan π_* can affect optimality with A* search, the probability that A* guided by h^α returns an optimal plan is at least $p_{\pi_*} = \alpha^{d_*}$ [Ernandes and Gori, 2004]. Since admissibility is a sufficient but not a necessary condition for optimally, p_{π^*} is a statistical lower bound for optimality.

Likely-admissibility is a weaker property than admissibility, but has the advantage that it is easier to obtain under a statistical machine learning approach as further discussed in Section 5.2.3.

5.2.3 Learning Heuristics from Existing Plans

Having access to prior knowledge in the form of a strong heuristic can greatly improve planning performance. However, crafting strong heuristics is a time-consuming process that requires domain-specific knowledge. An alternative approach is to learn heuristics from data. Previous research has shown that it is possible to learn a heuristic from some source tasks of a domain and then transfer the learned heuristic to some new task of the domain to produce good planning performance [Ernandes and Gori, 2004, Groshev et al., 2017].

The general framework for such a procedure is as follows: we are given a set of M plans $\Pi = \{\pi\}_{i=1}^{M}$ from a domain $\mathfrak{D}$. Given a plan $\pi = (s_0, s_1, s_2, ..., s_n = s_g) \in \Pi$ we compute for each $j < n$ the cost-to-goal from s_j to s_n, $y_j = \sum_{i=j}^{n-1} \mathcal{C}(s_i, s_{i+1})$. For each s_j we also compute a corresponding feature vector $x_j = F(s_j)$ where F is a function that converts a state to a feature representation that can be used as input to a supervised machine learning algorithm. Therefore, in totality, a training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ of size $N = \sum_{i=1}^{M} |\pi_i| - M$ is constructed by computing the feature vector / cost-to-goal pairs for the relevant states in each plan. Given $\mathcal{D}$, any number of regression-based supervised learning algorithms can be used to train a model $\hat{y}|x$ that learns to make a prediction $\hat{y}(x)$ that represents a mean cost-to-goal given some input x where $x = F(s)$ for any given state s.

Once trained, the model can be used as a heuristic by using $h(s) = \hat{y}(x)$. Since supervised learning algorithms have the ability to generalise to unseen data, this heuristic can be used on new, previously unseen, tasks from the given domain. In general, such a heuristic is not admissible and therefore can lead to non-optimal plans. The quality of the learned heuristic depends largely on the plans used for training - in-

tuitively, training on optimal or near-optimal plans will produce a heuristic that is a closer approximation of h^* and therefore have lower suboptimality.

Neural networks are a common choice of supervised learning algorithm for learning heuristic functions and have shown to produce good results on a variety of planning domains [Arfaee et al., 2010, Arfaee et al., 2011, Ernandes and Gori, 2004, Groshev et al., 2017]. We call heuristics based on neural networks *neural heuristics* in this chapter.

Learning heuristics from data is appealing but ideally, we want heuristics to be admissible so that they are guaranteed to produce optimal plans. As discussed in Section 5.2.2, a more feasible property to aim for with a machine learning approach is likely-admissibly. The notion of likely-admissibility was first introduced in conjunction with neural heuristics, where a novel cost function was developed to encourage admissibility when learning from existing plans [Ernandes and Gori, 2004]. This work showed that such neural heuristics, while not admissible, can produce strong planning performance with low suboptimality.[3]

5.2.4 Learning Heuristics by Bootstrapping

While learning heuristics from a dataset of existing plans as described in Section 5.2.3 is appealing, this approach is often infeasible. In particular, for domains with large state-spaces it is often computationally expensive to build an extensive dataset of plans. This problem is compounded by the fact that such a procedure produces the strongest heuristic when training on a dataset of optimal or near-optimal plans, which is typically more difficult to collect.

To overcome this, previous research has introduced a procedure for bootstrap learning of heuristics that interweaves planning and learning [Arfaee et al., 2010, Arfaee et al., 2011]. This procedure is shown in Algorithm 14. The premise of the algorithm is to start with some weak heuristic. This can be a completely uninformative heuristic, such as a neural network with random initial weights. The algorithm starts by generating *numTasks* random training tasks from a domain $\mathfrak{D}$ (line 1). A short time limit of t_{max} seconds is set, and a planner guided by the weak heuristic attempts to find a solution to each of these tasks within the time limit (line 8). Initially, most tasks are not solvable with the weak heuristic in this time limit. However, some of the easier tasks will be solved and their plans are added to a training dataset (lines 9-18). If not enough tasks are solved with the weak heuristic in the first iteration, as controlled by a parameter *minSolveToLearn*, the planning time limit is increased so

[3] More specifically, on the 15-puzzle domain, this approach produced optimal plans on 63.87% of tasks on a test set of size 700 with an average optimal cost-to-goal of 52.62.

that more tasks can be solved in the next iteration (line 26). This increase in time limit continues until enough tasks are solved with the weak heuristic. At that point the weak heuristic is updated with the plans of the solved tasks, thus becoming a stronger heuristic (lines 21-24). This process continues until all training tasks are solved and the algorithm then outputs a final heuristic (line 29).

Given a set of L admissible heuristics $\{h_i\}_{i=1}^{L}$, the bootstrap algorithm can be extended to incorporate this additional knowledge by using $h(s) = \max(\hat{y}(F(s)), h_{max}(s))$ for the heuristic, where $h_{max}(s) = \max(h_1(s), h_2(s), ..., h_L(s))$. Such an extension is useful as it gives the heuristic a jump-start in quality; otherwise the heuristic must be learned only from data during training. This technique is especially useful in domains with large state-spaces where learning from data alone becomes computationally expensive. Since the zero-heuristic is admissible for all planning domains, we can always use $h(s) = \max(\hat{y}(F(s)), 0)$.

The bootstrap learning approach has the advantage that it does not require a set of plans to be given upfront. However, the approach has two major shortcomings:

- The algorithm generates random training tasks from the given domain. This makes learning the heuristic inefficient because most tasks cannot be solved in the given time limit when the heuristic is still weak. As a result, much time is wasted during the planning phase of the algorithm.

  The authors who introduce the bootstrap learning approach propose an alternative way to generate training tasks by taking k random steps back from the goal. The idea under this approach is that initially k is set small so that the tasks generated in the first iteration are easily solved by the weak heuristic. The parameter k is then incremented by some fixed amount at each iteration, thus generating incrementally harder tasks to learn from. While this approach may improve efficiency, taking random steps to generate training tasks is still not ideal.

- In each iteration, the planner uses a heuristic that has been trained on non-optimal plans and then produces new non-optimal plans to learn from to make the heuristic stronger. As a result, errors compound with each iteration of the procedure and the final heuristic that the algorithm outputs produces high suboptimality.

5.2.5 Modelling Uncertainty

In this chapter we propose to overcome the shortcomings of the bootstrap learning approach discussed in Section 5.2.4 by incorporating measures of *epistemic* and

Algorithm 14: *Bootstrap*: bootstrap algorithm for learning heuristics.

Input: $\mathfrak{D}$, t_{max}, F, *minSolveToLearn*, *numTasks*

generate *numTasks* random tasks from $\mathfrak{D}$ and add them to a set $\mathfrak{T}$
initialise a set $\mathcal{D}\langle(F(\mathcal{S}), \mathbb{R}^+)\rangle$
initislaise a model $\hat{y}|x$
$h(s) \leftarrow \hat{y}(F(s))$ for all $s \in \mathcal{S}$
countSolved $\leftarrow 0$
while $|\mathfrak{T}| > 0$ **do**
    foreach $\mathcal{T} \in \mathfrak{T}$ **do**
        apply heurstic-based planner on $\mathcal{T}$ guided by h stopping early if no plan is found within t_{max} seconds
        if *plan* π *is found* **then**
            foreach $j \in [0 : |\pi| - 1]$ **do**
                $s_j \leftarrow$ state j in π
                $y_j \leftarrow$ cost-to-goal from s_j
                $x_j \leftarrow F(s_j)$
                $\mathcal{D} \cup \{(x_j, y_j)\}$
                remove $\mathcal{T}$ from $\mathfrak{T}$
                countSolved $\leftarrow$ *countSolved* $+ 1$
            end
        end
    end
    if *countSolved* $\geq$ *minSolveToLearn* **then**
        update model $\hat{y}|x$ with $\mathcal{D}$
        $h(s) \leftarrow \hat{y}(F(s))$ for all $s \in \mathcal{S}$
        $\mathcal{D} \leftarrow \phi$
        countSolved $\leftarrow 0$
    else
        $t_{max} \leftarrow 2 \times t_{max}$
    end
end
return h

aleatoric uncertainty into the bootstrap learning procedure.

Epistemic uncertainty accounts for uncertainty in the model of a process and can be explained away given enough data while aleatoric uncertainty accounts for uncertainty that the model cannot explain away due to probabilistic variation in the data.

As we describe in Section 5.3, epistemic uncertainty can be used to efficiently explore task-space so as to generate training tasks that are of the right level - that is, the tasks are easy enough to solve while being difficult enough for learning to progress. Meanwhile, epistemic and aleatoric uncertainty are combined so that the heuristic remains likely-admissible during planning and this mitigates the compounding error problem that results in high suboptimality. In order to achieve this, we need to enrich our model $\hat{y}|x$ to produce metrics of these uncertainties. One way to achieve this is through a Bayesian approach.

As an illustrative example to explain the concepts of epistemic and aleatoric uncertainty, consider the process of tossing a biased coin. This process has outcome $z = 1$ with probability θ if the coin lands heads, and $z = 0$ with probability $1 - \theta$ if the coin lands tails. The process can be modelled with a Bernoulli distribution $z|\theta \sim Bern(\theta)$.

Suppose that we do not know θ but we can toss the coin to learn about the process. With a Bayesian approach, we start out with a prior belief over θ having observed no data. This prior belief is captured with a distribution $P(\theta)$ that describes our beliefs over possible values of θ. Once data $\mathcal{D}$ is collected, we update our beliefs over θ using Bayes rules [Gelman et al., 1995]:

$$P(\theta|\mathcal{D}) = \frac{P(\mathcal{D}|\theta)P(\theta)}{P(\mathcal{D})} \propto P(\mathcal{D}|\theta)P(\theta) \tag{5.1}$$

where $P(\mathcal{D}|\theta)$ is called the likelihood and $P(\mathcal{D})$ is called the marginal likelihood. The marginal likelihood is a constant relative to θ, hence leading to the proportionality relationship in equation 5.1. The distribution $P(\theta|\mathcal{D})$ is called the posterior distribution and captures our new beliefs over θ that take the observed data into account.

Given $P(\theta|\mathcal{D})$ we can then infer the posterior predictive distribution over z

$$P(z|\mathcal{D}) = \int_{\theta} P(z|\theta)P(\theta|\mathcal{D})d\theta$$

that allows us to model the outcome of the process under our updated beliefs as

captured by the posterior distribution.

To make the Bayesian approach more concrete, consider a Bernoulli distribution under a prior belief that $\theta \sim Beta(1, 1)$ follows a uniform Beta distribution. It can be shown that this setup produces a posterior predictive distribution

$$z|N_1, N_0 \sim Bern\left(\frac{N_1 + 1}{N_1 + N_0 + 2}\right)$$

where N_1 and N_0 are the observed heads and tails counts when flipping the coin respectively [Murphy, 2012].

Suppose $\theta = 0.8$ is the true, and unknown, probability of landing heads. It is known that the variance of a Bernoulli distribution with probability $\tilde{\theta}$ is $\tilde{\theta}(1 - \tilde{\theta})$. Therefore, when we have not observed any data - i.e. $N_1 = N_2 = 0$ - the variance of the posterior predictive is $0.5 \times 0.5 = 0.25$. This is higher than the variance of the true process, $0.8 \times 0.2 = 0.16$. The reason for this is because the variance of the posterior predictive distribution takes into account both the uncertainty of the coin toss outcome as well as the uncertainty over the model parameter θ.

Now, suppose we flip the coin 100 times and observe $N_1 = 75$ and $N_0 = 25$. We now have that the posterior predictive distribution has variance $\frac{76}{102} \times \frac{26}{102} = 0.19$. Therefore, by observing data from the process we have reduced the uncertainty and shifted closer to the variance of the true distribution.

In this example, epistemic uncertainty arises because of the unknown value of the model parameter θ. Given enough data we can reduce the epistemic uncertainty within arbitrary precision. Meanwhile the uncertainty because of probabilistic variation when modelling the process is the aleatoric uncertainty, and this cannot be explained away no matter how much data is collected. That is, even if we converge to $\theta = 0.8$ by collecting lots of data from flipping the coin, the posterior predictive will still have variance $0.8 \times 0.2 = 0.16$. Aleatoric uncertainty can only be explained away by adding explanatory variables to the data collected when observing the process and incorporating this new information into the model.[4]

A Bayesian modelling approach is very powerful allowing, amongst other benefits, the ability to produce measures of uncertainty. Unfortunately, Bayesian inference is often intractable, largely because of the need to compute the marginal likelihood

[4]While it is unclear how to do this in the case of coin tossing, there are many applications where adding additional explanatory variables to a model can reduce aleatoric uncertainty. A simple example is the task of predicting an individual's mass. Intuitively, a model that uses both height and age as explanatory variables will produce lower aleatoric uncertainty over a model that only uses one of these.

$$P(\mathcal{D}) = \int_{\theta} P(\mathcal{D}|\theta)P(\theta)d\theta.$$

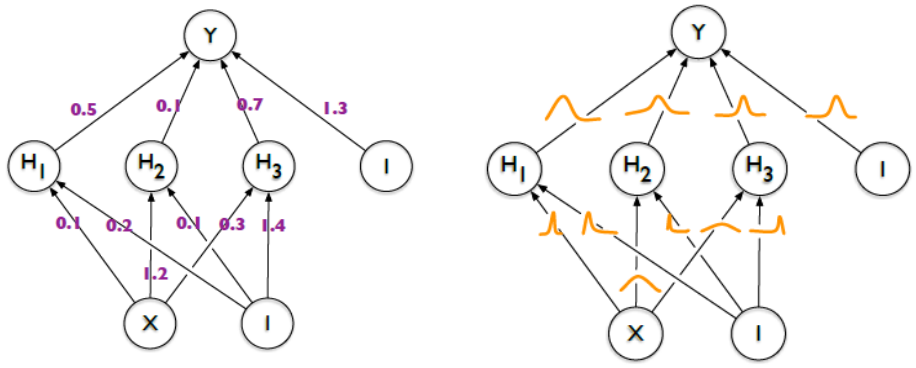
5.2.6 Bayesian Neural Networks

In Section 5.2.5 we illustrate the concepts of epistemic and aleatoric uncertainty by using Bayesian methods for a simple coin tossing process. However, these ideas extend to more complex processes that require more sophisticated models. For example, Bayesian linear regression enriches standard linear regression to produce a full posterior predictive distribution over the target variable y, rather than only a mean point estimate as in the standard setting. It is possible to show that with the appropriate choice of prior and likelihood, inference under Bayesian linear regression is tractable and measures of epistemic and aleatoric uncertainty can be extracted from the resulting posterior predictive distribution [Murphy, 2012].

As discussed in Section 5.2.3 neural networks have been shown to produce state-of-the-art performance on the task of learning heuristics from data. Neural network models are an extension of linear regression models that operate by taking in some input and passing it through multiple hidden layers made up of nodes called neurons, and applying a (non-linear) transformation at each such layer. Standard neural network models use fixed point weights between connecting neurons, as shown in Figure 5.1a. Once the network architecture is defined, neural networks are most commonly trained with the well-known backpropagation Algorithm [Rumelhart et al., 1986].

Given the success of standard neural networks in solving a variety of machine learning tasks, it is natural to want to extend them to the Bayesian setting in the same way that Bayesian linear regression extends standard linear regression. Bayesian neural networks are straightforward to define by simply representing the connections in the network as probability distributions over the weights rather than fixed point values, as shown in Figure 5.1b. Unfortunately, while easy to define, Bayesian inference is generally intractable with neural network models because of the large number of parameters and functional form of these models [Shridhar et al., 2019]. As a result, we must resort to approximate methods to estimate the posterior predictive distribution.

While various Bayesian neural network approximations exist [Gal and Ghahramani, 2016, Hernandez-Lobato and Adams, 2015, Sun et al., 2017, Blundell et al., 2015], in this chapter we use a *weight uncertainty neural network* (WUNN) [Blundell et al., 2015] with independent Gaussian weights to measure epistemic uncertainty for our practical implementation discussed in Section 5.4. We furthermore extend the neural network with an additional output neuron to measure aleatoric uncertainty under



(a) Standard neural network model with weights represented as fixed point values.

(b) Bayesian neural network with weights represented as probability distributions.

Figure 5.1: Comparison between standard and Bayesian neural networks (image source: [Blundell et al., 2015]).

the assumption of a Gaussian response [Kendall and Gal, 2017]. Our choice of approximation is motivated by a combination of simplicity of implementation together with robust performance on experimental domains when testing possible solutions. In the remainder of this section we describe this model in more detail.

A neural network can be viewed as a probabilistic model $P(y|x, \mathbf{w})$ [Murphy, 2012] where x is the input to the network and $\mathbf{w}$ are the weights. For regression it is most common to assume that $y|x, \mathbf{w} \in \mathbb{R}$ follows a Gaussian distribution so that $y|x, \mathbf{w} \sim \mathcal{N}(\hat{y}(x, \mathbf{w}), \sigma_a^2(x, \mathbf{w}))$ where $\hat{y}$ is the mean and σ_a^2 is the variance of the distribution. Typically when doing regression with neural networks only a single output neuron is used to learn $\hat{y}$ while σ_a^2 is assumed to be a known constant, and this corresponds to the well-known squared loss minimisation objective [Murphy, 2012]:

$$\mathcal{L}(\mathcal{D}, \mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}(x_i, \mathbf{w}) - y_i)^2. \tag{5.2}$$

However, it is straightforward to adjust the network to have two output neurons and learn σ_a^2 as well. This leads to the following minimisation objective function [Kendall and Gal, 2017]:

$$\mathcal{L}(\mathcal{D}, \mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \frac{(\hat{y}(x_i, \mathbf{w}) - y_i)^2}{2\sigma_a^2(x_i, \mathbf{w})} + \frac{1}{2} \log \sigma_a^2(x_i, \mathbf{w}), \tag{5.3}$$

From an optimisation perspective, the intuition behind equation 5.3 is as follows: with the squared loss objective in equation 5.2 the only way to reduce the loss for

some training data point (x_i, y_i) is to shift $\hat{y}(x_i, \mathbf{w})$ closer to y_i by adjusting $\mathbf{w}$. Now consider equation 5.3. The first term in this equation can be reduced in one of two ways: either by shifting $\hat{y}(x_i, \mathbf{w})$ closer to y_i, while fixing $\sigma_a^2(x_i, \mathbf{w})$; or by increasing $\sigma_a^2(x_i, \mathbf{w})$, while fixing $\hat{y}(x_i, \mathbf{w})$. If the loss was just this first term then the model could learn to minimise this objective by simply making $\sigma_a^2(x_i, \mathbf{w})$ very large, which would lead to poor predictive performance. To counter this, the second term in equation 5.2 acts as a regulariser that penalises large values of $\sigma_a^2(x_i, \mathbf{w})$.

Since the output of a neural network is unbounded, in practice we learn a second output neuron $\zeta \in \mathbb{R}$ and then apply a transformation to get $\sigma_a \in \mathbb{R}^+$ such as $\sigma_a = \log(1 + \exp(\zeta))$. Then σ_a^2 captures the noise in the data and is thus a measure of aleatoric uncertainty.

Epistemic uncertainty is more difficult to model as it requires the posterior distribution $P(\mathbf{w}|\mathcal{D})$. WUNNs are an effective and computationally efficient method to approximate this posterior distribution [Blundell et al., 2015]. WUNNs adjust the standard neural network model, that uses fixed point weights, and instead represent each weight as a parameterised distribution. So long as the individual weight distributions are independent of each other, WUNNs are flexible with regards to the choice of distribution used. In our implementation we use a Gaussian for each weight, $w_i|\mathcal{D} \sim \mathcal{N}(\mu_{w_i|\mathcal{D}}, \sigma^2_{w_i|\mathcal{D}})$ with a Gaussian prior $w_i \sim \mathcal{N}(\mu_0, \sigma_0^2)$ where $\mu_{w_i|\mathcal{D}}$ and μ_0 are the mean values of the posterior and prior respectively, while $\sigma^2_{w_i|\mathcal{D}}$ and σ_0^2 are the variance values of the posterior and prior respectively.

WUNNs operate by minimising the variational approximation to the posterior:

$$\mathcal{L}(\mathcal{D}, \theta) = \beta KL[P(\mathbf{w}|\theta)||P(\mathbf{w})] - \mathbb{E}_{P(\mathbf{w}|\theta)}[\log P(\mathcal{D}|\mathbf{w})], \tag{5.4}$$

where KL is the Kullback-Leibler divergence, $\theta = \{(\mu_{w_i|\mathcal{D}}, \sigma^2_{w_i|\mathcal{D}})\}_i$ for each weight connection in the network and $\beta > 0$ controls how much weight to give the prior relative to the likelihood [Higgins et al., 2017]. The second term in equation 5.4 maximises the likelihood relative to the training data, while the first term acts as a regulariser that encourages the posterior to remain close to the prior. A common strategy with regards to the parameter β is to start with a large value of β and decay it over training iterations. This strategy initially gives more emphasis to the prior and gradually shifts focus to the likelihood as the network is trained on more data.

In practice, this objective is approximated with Monte Carlo simulation during training by sampling $z_i^s \sim \mathcal{N}(0, 1)$ for each weight connection in the network and then applying the transformation $w_i^s = z_i^s \sigma_{w_i|\mathcal{D}} + \mu_{w_i|\mathcal{D}}$ where $s \in [1, S]$ and S is the

number of Monte Carlo simulations averaged over.

Given a WUNN trained to minimise this objective, we can obtain a measure of epistemic uncertainty for an input x by sampling weights from the network and computing the variance of the network output

$$\sigma_e^2(x) = \frac{1}{K}\sum_{k=1}^{K} \hat{y}^2(x, \mathbf{w}_k) - \hat{y}^2(x),$$

where K is the number of samples from the network and the mean is computed as

$$\hat{y}(x) = \frac{1}{K}\sum_{k=1}^{K} \hat{y}(x, \mathbf{w}_k).$$

The intuition behind WUNNs is as follows: when the network is trained, it is required to fit the parameters of the network to both the observed training data *and the prior*. Therefore, when a test input x is passed through the network that is similar to the data observed during training, the model makes a prediction with low epistemic uncertainty. This is because, for this type of input, the model would have set the gaussian weights during training in such a way so as to produce a consistent prediction, and this is only possible if σ_e^2 is low. However, if x is different from the data observed during training, then the model makes a prediction that tends towards the prior. Therefore, by setting a high prior variance for the weights we can expose observations that are dissimilar to what was observed during training, as the model will return large values of σ_e^2 given this type of input.

Finally, given an input x we can approximate the posterior predictive distribution as

$$y|x \sim \mathcal{N}(\hat{y}(x), \sigma_t^2(x)), \tag{5.5}$$

where $\sigma_t^2(x) = \sigma_a^2(x) + \sigma_e^2(x)$ and $\sigma_a^2(x) = \frac{1}{K}\sum_{k=1}^{K} \sigma_a^2(x, \mathbf{w}_k)$.[5]

5.3 Conceptual Framework

In this section we outline a conceptual framework for efficient learning of likely-admissible heuristics via a bootstrap approach. This framework overcomes the shortcomings of previous bootstrap approaches that are sample inefficient and produce

[5]For clarity, the subscripts t, a and e in these equations stand for 'total', 'aleatoric' and 'epistemic' respectively.

heuristics with high suboptimality as further discussed in Section 5.2.4. Our approach is based on utilising epistemic uncertainty to efficiently explore task-space, while combining epistemic and aleatoric uncertainty to ensure that the heuristic remains likely-admissible during planning so that the final heuristic produces low suboptimality.

In Section 5.2.6 we explain how these uncertainty measures can be obtained using a Bayesian neural network model. While we use this concrete implementation when running experiments in Section 5.5, in this section we outline a conceptual framework that can be used in conjunction with a broad range of models. Specifically, our framework assumes that we have a model $\mathfrak{M}$ of the form

$$y|x \sim \mathbb{D}(\mu(x), \nu_a(x) + \nu_e(x)), \tag{5.6}$$

where $\mathbb{D}$ is a distribution with mean parameter μ, aleatoric uncertainty parameter ν_a and epistemic uncertainty parameter ν_e.

In the context of learning heuristics from data, epistemic uncertainty arises because a state s may induce features $x = F(s)$ that are dissimilar to what was observed during training, while aleatoric uncertainty arises because either 1) the features extracted from F are insufficient to deterministically explain the cost-to-goal observations; or 2) we train on non-optimal plans so identical states can map to different cost-to-goal observations.

It is important to note that under our framework, there is no fixed heuristic value for a given state. Rather, we have a distribution of values as demonstrated in Figure 5.2, where in this example $\mathbb{D}$ is a Gaussian distribution. When planning, a specific heuristic value must then be chosen from this distribution. Under our framework, and as we demonstrate experimentally in Section 5.5, there is a trade-off based on the value chosen. A more conservative choice encourages a decrease in suboptimality during planning at the cost of an increase in the number of nodes expanded (and therefore planning time). Meanwhile, a less conservative choice encourages an increase in suboptimality at the cost of a decrease in the number of nodes expanded during planning.

In Section 5.2.2 we discuss the notion of likely-admissible heuristics. While weaker than admissible heuristics that guarantee optimal plans, likely-admissible heuristics are admissible with some probability and therefore produce likely-optimal plans. Likely-admissibility can formally be described under our proposed framework by using a value y^α such that $P(y^\alpha \leq y|x) = \alpha$ where $\alpha \in (0, 1)$ for the heuristic. The value y^α can be computed from the inverse cumulative distribution function of $\mathbb{D}$

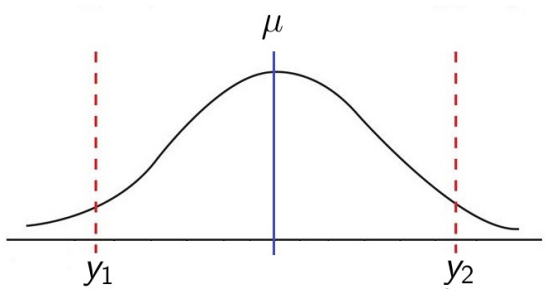


Figure 5.2: Two possible heuristic values, y_1 and y_2, that can be chosen from $\mathbb{D}$. y_1 is more conservative than y_2.

which is done either analytically or using approximation methods depending on the form of $\mathbb{D}$. Choosing a high value for α maps to a more conservative heuristic value that is closer to the left tail of $\mathbb{D}$. We note that for this formalisation of likely-admissibility to hold we require that $\mathfrak{M}$ learns from training data of optimal plans so that $y|F(s)$ is a model for $h^*(s)$.

Combining these ideas, we introduce two algorithms: *GenerateTask* (Algorithm 15) and *LearnHeuristic* (Algorithm 16). The Algorithm *GenerateTask* is used to generate a training task that is then solved with *LearnHeuristic*.

The key idea for *GenerateTask* is to use epistemic uncertainty to generate a training task that is easy enough to solve while being difficult enough so that learning from the solved task makes the heuristic stronger. To achieve this, we start at the goal state s_g (line 2). With some user specified value ϵ, we take steps back from the goal state until we observe a state s such that $\nu_e(F(s)) \geq \epsilon$ (line 13). We then stop execution and build a task $\mathcal{T}$ with start state $s_0 = s$ (lines 14-15). The algorithm is designed to efficiently explore task-space so as to seek states with high epistemic uncertainty by sampling states from the softmax distribution derived from the epistemic uncertainties of all possible next states (line 12). As a result, our algorithm covers the state-space more efficiently than using random task generation.

Since the algorithm builds tasks by taking steps backward from the goal we further need access to an undo effect function $\mathcal{E}^{\text{undo}}(s')$ that returns the set of all states s that move to s' under some operator in $\Omega(s)$, i.e. $\mathcal{E}^{\text{undo}}(s') = \{s \mid \exists o \in \Omega(s)$ such that $\mathcal{E}(s,o) = s'$ where $s \in \mathcal{S}\}$. This definition of an undo function can be used even in domains with non-reversable actions. An example of such a domain is Sokoban, where the agent can push a box into a deadlock state s' from which the task can no longer be solved. In this scenario, $\mathcal{E}^{\text{undo}}(s')$ will contain a state that undoes the deadlock. However, if the domain has reversable actions, as is the case of the domains used in this chapter, we may simply use $\mathcal{E}^{\text{undo}}(s') = \mathcal{E}^{rev}(s') = \{\mathcal{E}(s',o) \mid o \in \Omega(s')\}$.

Algorithm 15: *GenerateTask*: generate task using epistemic uncertainty.

Input: $\mathfrak{D}$, $\mathfrak{M}$, ϵ

1 $\mathcal{S} \leftarrow$ state-space of $\mathfrak{D}$
2 $s' \leftarrow$ goal state of $\mathfrak{D}$
3 initialise a set $visited\langle\mathcal{S}\rangle$
4 **repeat**
5 $\quad visited \cup \{s'\}$
6 $\quad$ initialise a set $states\langle(\mathcal{S}, \mathbb{R}^+)\rangle$
7 $\quad$ **foreach** $s \in \mathcal{E}^{undo}(s')$ **do**
8 $\quad\quad x \leftarrow F(s)$
9 $\quad\quad$ compute $\nu_e(x)$ from $\mathfrak{M}$
10 $\quad\quad states \cup \{(s, \nu_e(x))\}$
11 $\quad$ **end**
12 $\quad$ sample from softmax distribution derived from the epistemic uncertainties in $states$ to obtain some pair $(s, \nu_e(x))$
13 $\quad$ **if** $\nu_e(x) \geq \epsilon$ *or* $visited = \mathcal{S}$ **then**
14 $\quad\quad \mathcal{T} \leftarrow \langle \mathcal{S}, \Omega, \mathcal{E}, \mathcal{C}, s, s_g \rangle$ where the components $\mathcal{S}$, Ω, $\mathcal{E}$, $\mathcal{C}$ and s_g come from $\mathfrak{D}$
15 $\quad\quad$ return $\mathcal{T}$
16 $\quad$ **end**
17 $\quad s' \leftarrow s$
18 **until** *forever*

The next step is to learn the heuristic. This is achieved with the Algorithm *Learn-Heuristic*. The algorithm proceeds to generate *NumTasksPerIter* tasks with *GenerateTask* then solve each task using a user specified admissibility probability α (lines 5-6). The plan of each solved task is added to a training dataset (lines 7-12). Solving the tasks with y^α ensures that plans that the heuristic is trained on are encouraged to be likely-optimal and, as we show empirically in Section 5.5, this mitigates the compounding error problem that exists with previous bootstrap approaches. We note that in our algorithm the training data consists of non-optimal plans and so y^α is not formally a likely-admissible heuristic. Nevertheless, we make the assumption that the plans are close enough to optimal that any errors incurred are negligible - in practice the larger α is set, the closer this assumption holds. After the *NumTasksPerIter* tasks are solved the heuristic is updated with the training dataset, thus becoming a stronger heuristic (line 14). A subtle but important requirement for the algorithm is that when we train the heuristic, the epistemic uncertainty on all entries in the training dataset must fall below ϵ (line 14). This is important to ensure that the next time *GenerateTask* is called, we generate a task that is different to what has previously been trained on.

To provide a better intuition for how the algorithms operate, we provide a visual illustration for the 15-puzzle. Consider Figure 5.3. The first step in Figure 5.3a shows the starting point for the algorithm which is the goal state. We sample from the softmax distribution derived from the epistemic uncertainties of the possible

Algorithm 16: *LearnHeuristic*: learn a likely-admissible heuristic.

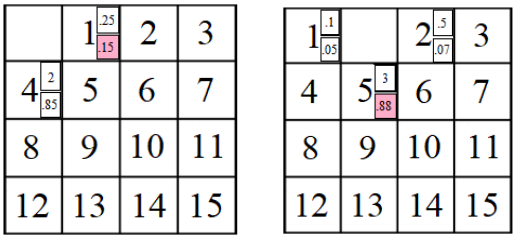
Input: $\mathfrak{D}$, F, *NumTasksPerIter*, α, ϵ

initialise a model $\mathfrak{M}$ as per equation 5.6
initialise a set $\mathcal{D}\langle(F(\mathcal{S}), \mathbb{R}^+)\rangle$
repeat
  for $i \in [1 : \mathit{NumTasksPerIter}]$ **do**
    $\mathcal{T} \leftarrow \mathit{GenerateTask}(\mathfrak{D}, \mathfrak{M}, \epsilon)$
    solve $\mathcal{T}$ using $\max(y^\alpha, 0)$ as the heuristic to obtain a plan π
    foreach $j \in [0 : |\pi| - 1]$ **do**
      $s_j \leftarrow$ state j in π
      $y_j \leftarrow$ cost-to-goal from s_j
      $x_j \leftarrow F(s_j)$
      $\mathcal{D} \cup \{(x_j, y_j)\}$
    end
  end
  train $\mathfrak{M}$ from $\mathcal{D}$ until for all entries (x_i, y_i) we have $\nu_e(x_i) < \epsilon$ and the training error $\sum_i (\mu(x_i) - y_i)^2$ is sufficiently small
until *forever*

previous states. We were more likely to take a step with the DOWN operator but we happened to sample the RIGHT operator. Since the RIGHT operator leads to a state with $\nu_e = 0.25$ which is less than $\epsilon = 1$ we continue to run *GenerateTask*. In the second step shown in Figure 5.3b we again sample from the softmax distribution and this time we sample the DOWN operator which leads to a state with $\nu_e = 3$. Since this value is greater than $\epsilon = 1$ we stop *GenerateTask*. Figure 5.3c then shows the start state that is used to build the training task to be solved by the heuristic. In this process we are more likely to sample states with high epistemic uncertainly and this makes exploration in task-space more efficient than random task generation.

When running *LearnHeuristic* we solve this task and add the plan to the training dataset. Thereafter, we train the heuristic until $\nu_e < \epsilon$ for all states in the training dataset. Suppose in the next iteration of the algorithm we generate a task that takes the same trajectory as in Figure 5.3. As shown in Figure 5.4 we will generate a more complex task because in the previous run of training the heuristic would have reduced the epistemic uncertainty of the state in Figure 5.3c below ϵ. This incremental increase in complexity of tasks makes the heuristic stronger with each iteration of the procedure.

In practice you will not run *LearnHeuristic* forever, but will cut-off training once the heuristic has been trained on enough tasks. One way to gauge a good cut-off point is to monitor the number of steps taken when calling *GenerateTask*. At that point, the heuristic can be transferred to new test tasks of the domain. The parameter α can then be set to trade-off between suboptimality and planning time. This ability to tune α is an advantage of our proposed framework as the designer can decide,



(a) First step. (b) Second step.

1	5	2	3
4		6	7
8	9	10	11
12	13	14	15

(c) Start state that is used to generate the task.

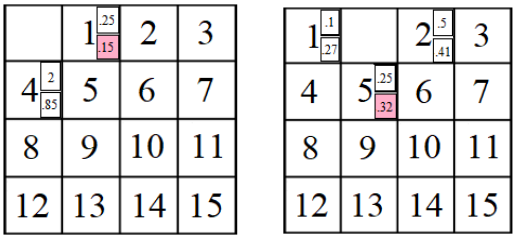
Figure 5.3: Hypothetical simulation of *GenerateTask* with $\epsilon = 1$ for the 15-puzzle. The blocks on the top right of a tile represent the epistemic uncertainty. The blocks on the bottom right of a tile represent the derived softmax distribution, with the pink block being the sampled operator.

based on their application, which takes priority.

The algorithms introduced in this section take their inspiration from the R_{max} Algorithm [Brafman and Tennenholtz, 2002] that was discussed in Section 4.2.3. The R_{max} algorithm is an efficient model-based RL algorithm that operates under the *optimism in the face of uncertainty* principle. The core idea with R_{max} is to drive exploration by assuming that unexplored parts of an environment produce the highest possible payoff. This encourages exploration to these parts, at which point the ground truth is learned and different unexplored parts can then be prioritised. The *GenerateTask* algorithm operates under a similar principle. The algorithm drives exploration to unexplored states with high epistemic uncertainty. These states are then used as start states for generating training tasks. In *LearnHeuristic*, after tasks are generated with *GenerateTask*, the tasks are then solved and the heuristic learns from their plans. Therefore, the next time *GenerateTask* is called it prioritises tasks that are different from those that have been previously generated.

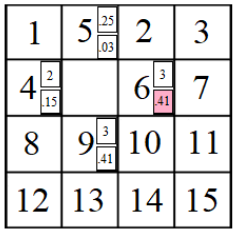
5.4 Practical Implementation

In Section 5.3 we describe a conceptual framework to efficiently learn likely-admissible heuristic that works with a broad range of models that produce measures

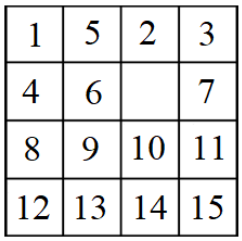


(a) First step.

(b) Second step.



(c) Third step.



(d) Start state that is used to generate the task.

Figure 5.4: Hypothetical simulation of *GenerateTask* after training on the task that was generated with start state as in Figure 5.3c.

of epistemic and aleatoric uncertainty. In this section we instantiate this conceptual framework with the WUNN model described in Section 5.2.6 where $\mu = \hat{y}$, $\nu_a = \sigma_a^2$ and $\nu_e = \sigma_e^2$. However, to get an algorithm that works in practice with this WUNN setup requires additional considerations. In this section we present two additional algorithms, *GenerateTaskPrac* (Algorithm 17) and *LearnHeuristicPrac* (Algorithm 18) that outline and implement these considerations.

The general Algorithm *GenerateTask* requires us to store every visited state, however, this becomes restrictive if the state-space is large. An alternative approach is to include an additional parameter *MaxSteps* and let the algorithm terminate if this number of steps has been taken (Algorithm 17, line 17). Setting *MaxSteps* requires domain knowledge, as it needs to be set larger than the optimal number steps required to solve the tasks of the domain. A second modification, introduced previously [Arfaee et al., 2010, Arfaee et al., 2011], is to exclude at each step the state that would take you back to the previously observed state (Algorithm 17, lines 9-11). This encourages the algorithm to move further from the goal state with every step. Lastly, we use $\mathcal{E}^{rev}$ in our implementation as all our experimental domains have reversable actions (Algorithm 17, line 8).

The general Algorithm *LearnHeuristic* also requires various considerations. For planning it is advantageous to expand as many nodes per second as possible, but working with WUNNs requires multiple passes through the network to get good

Algorithm 17: *GenerateTaskPrac*: practical implementation of *GenerateTask*.

Input: $\mathfrak{D}$, nn_{WUNN}, ϵ, *MaxSteps*, K

$\mathcal{S} \leftarrow$ state-space of $\mathfrak{D}$
$s' \leftarrow$ goal state of $\mathfrak{D}$
$numSteps = 0$
$s'' \leftarrow NULL$
repeat
  $numSteps \leftarrow numSteps + 1$
  initialise a dictionary $states\langle \mathcal{S}, \mathbb{R}^+ \rangle$
  foreach $s \in \mathcal{E}^{rev}(s')$ **do**
    if $s'' \neq NULL$ *and* $s'' = s$ **then**
      continue loop
    end
    $x \leftarrow F(s)$
    compute $\sigma_e^2(x)$ from nn_{WUNN} using K samples
    $states[s] \leftarrow \sigma_e(x)$
  end
  sample from softmax distribution derived from *states.Values* to obtain some pair $(s, \sigma_e(x))$
  if $\sigma_e^2(x) \geq \epsilon$ *or* $numSteps \geq MaxSteps$ **then**
    $\mathcal{T} \leftarrow \langle \mathcal{S}, \Omega, \mathcal{E}, \mathcal{C}, s, s_g \rangle$ where the components $\mathcal{S}$, Ω, $\mathcal{E}$, $\mathcal{C}$ and s_g come from $\mathfrak{D}$
    return $\mathcal{T}$
  end
  $s'' \leftarrow s'$
  $s' \leftarrow s$
until *forever*

estimates of the cost-to-goal and this is slow. A simple solution is to use two separate neural networks: one WUNN that learns σ_e^2 (Algorithm 18, line 1) and is only used for generating tasks and one standard feed-forward neural network (FFNN) that learns $\hat{y}$ and σ_a^2 and is only used for planning (Algorithm 18, line 2).

Unfortunately, for planning under our framework we still need σ_e^2 to compute σ_t^2. Furthermore, σ_a^2 is only reliable when run on data that is similar to what the heuristic has already been trained on, and can otherwise be highly erroneous. Therefore, we found the following approximation to work well: in our algorithm we maintain a training dataset with observed features / cost-to-goal pairs (Algorithm 18, line 8). From this, we keep track of y^q, the q^{th} quantile of the cost-to-goal values at each iteration (Algorithm 18, line 34). When planning, if $\hat{y} < y^q$ use $\sigma_t^2 = \sigma_a^2$ otherwise use $\sigma_t^2 = \epsilon$ (Algorithm 18, line 12). The intuition behind this approximation is that if we are in a region of the feature-space where the mean value is less than a cost-to-goal value that was previously trained on, then we can assume σ_e^2 is negligible and just use σ_a^2. Otherwise, we use ϵ as a lower bound for σ_e^2 and disregard σ_a^2 because we may be in a region of the feature-space where this measure is inaccurate. Since we now have $\sigma_t^2 \leq \sigma_a^2 + \sigma_e^2$, this approximation has the effect of producing a heuristic that has lower admissibility probability for a given value of α.

A second consideration is the handling of β in equation 5.4. If β is too large then it impossible to achieve the requirement in *LearnHeuristic* line 14 because we cannot decrease the epistemic uncertainty sufficiently. However, if we make β too small then we generate tasks that are too complex for the heuristic to solve quickly. Coupled with this, just ensuring $\sigma_e^2 < \epsilon$ is not sufficient because WUNNs obtain an empirical estimate of the epistemic uncertainty which contains sampling errors. Therefore, it is prudent to ensure that $\sigma_e^2 < \kappa\epsilon$ where $\kappa \in (0, 1)$. In summary our approach is as follows: start with some initial value for β, say $\beta := \beta_0$ (Algorithm 18, line 6). Train for some maximum number of iterations *MaxTrainIter* and at each iteration sample a mini-batch of size *MiniBatchSize* from the training data (Algorithm 18, line 33). After each training iteration, if $\sigma_e^2 < \kappa\epsilon$ for all entries in the entire training dataset then stop training early, otherwise complete training for *MaxTrainIter* iterations and then set $\beta := \eta\beta$ where $\eta \in (0, 1)$. This criterion ensures that we either reduce the epistemic uncertainty on the training data sufficiently or reduce the importance of the prior in the next iteration.

Next we consider the choice of α. We want α to be as large as possible so as to encourage admissibility. However, if α is too large then the heuristic can grossly underestimate the optimal cost-to-goal and this leads to more nodes being generated during planning - in fact, this can make planning infeasible. To mitigate this, we first introduce a time limit t_{max} so that if the planner cannot solve a task in

Algorithm 18: *LearnHeuristicPrac*: practical implementation of *LearnHeuristic*.

Input: $\mathfrak{D}$, F, *NumIter*, *NumTasksPerIter*, *NumTasksPerIterThresh*, α_0, Δ, ϵ, β_0, η, κ, ϵ, *MaxSteps*, *MemoryBufferMaxRecords*, *TrainIter*, *MaxTrainIter*, *MiniBatchSize*, t_{max}, μ_0, σ_0^2, q, K

initialise a WUNN nn_{WUNN} using priors μ_0 and σ_0^2
initialise a FFNN nn_{FFNN}
initialise a list $memoryBuffer\langle(F(\mathcal{S}), \mathbb{R}^+)\rangle$
$y^q \leftarrow -\infty$
$\alpha \leftarrow \alpha_0$
$\beta \leftarrow \beta_0$
define a function $h(\alpha, \mu, \sigma)$ that that returns y^α where $P(y^\alpha \leq y \mid y \sim \mathcal{N}(\mu, \sigma^2)) = \alpha$
for $n \in [1 : NumIter]$ **do**
    $numSolved \leftarrow 0$
    for $i \in [1 : NumTasksPerIter]$ **do**
        $\mathcal{T} \leftarrow GenerateTaskPrac(\mathfrak{D}, nn_{WUNN}, \epsilon, MaxSteps, K)$
        try solve $\mathcal{T}$ within t_{max} seconds using a heuristic-based planner with $\max(h, 0)$ as the heuristic to obtain a plan π; when planning with h for each state visited s, pass α, $\hat{y}(x)$ and $\sigma_t^2(x)$ as the parameters respectively where $\sigma_t^2(x) \leftarrow \sigma_a^2(x)$ if $\hat{y}(x) < y^q$ else $\sigma_t^2(x) \leftarrow \epsilon$ and where $x \leftarrow F(s)$
        if *plan* π *was found* **then**
            $numSolved \leftarrow numSolved + 1$
            foreach $j \in [0 : |\pi| - 1]$ **do**
                $s_j \leftarrow$ state j in π
                $y_j \leftarrow$ cost-to-goal from s_j
                $x_j \leftarrow F(s_j)$
                $memoryBuffer.Add((x_j, y_j))$
            end
        end
    end
    trim *memoryBuffer* to keep the most recently added $MemoryBufferMaxRecords$ records
    $updateBeta \leftarrow 1$
    if $numSolved < NumTasksPerIterThresh$ **then**
        $\alpha_{prev} \leftarrow \alpha$
        $\alpha \leftarrow \max(\alpha - \Delta, 0.5)$
        if $\alpha_{prev} \neq \alpha$ **then**
            $updateBeta \leftarrow 0$
        end
    end
    train nn_{FFNN} using entire *memoryBuffer* for *TrainIter* iterations
    train nn_{WUNN} from *memoryBuffer* for *MaxTrainIter* iterations using a minibatch size of *MiniBatchSize* per iteration; if after any iteration $\sigma_e^2(x_i) < \kappa\epsilon$ for all (x_i, y_i) in *MemoryBuffer* then stop early else complete *MaxTrainIter* iterations and if $updateBeta = 1$ then $\beta \leftarrow \eta\beta$
    $y^q \leftarrow$ quantile q of the cost-to-goal values in *memoryBuffer*
end

t_{max} seconds we stop planning. We start with an initial value for α, say $\alpha := \alpha_0$ (Algorithm 18, line 5), and we try to solve all *NumTasksPerIter* tasks with y^α within t_{max} seconds (Algorithm 18, line 12). We keep track of how many tasks are solved and if the number of solved tasks in an iteration is less than some threshold *NumTasksPerIterThresh* we update $\alpha := \max(\alpha - \Delta, 0.5)$ where $\Delta > 0$ (Algorithm 18, line 27). The idea behind this update is that if we cannot solve enough tasks in an iteration then we should sacrifice admissibility so that we can make learning progress. We floor the value of α at 0.5 which then reverts to the mean value $\hat{y}$.

Furthermore, we implement a rule that if α is updated in an iteration of *LearnHeuristicPrac* then β is not updated when training the WUNN. This is because if α is decreased, then it makes sense to leave β unchanged and try solve tasks with this lower admissibility probability in the next iteration - this rule is controlled through a Boolean variable *updateBeta* that is set to 0 if α is updated in an iteration (Algorithm 18, line 29).

Additional considerations include 1) we must cut-off *LearnHeuristicPrac* after some *NumIter* number of iterations (Algorithm 18, line 8); 2) Instead of a set $\mathcal{D}$ we use a list *MemoryBuffer* for the training data and we store only the most recent *MemoryBufferMaxRecords* added to the list (Algorithm 18, line 23); 3) when calling *GenerateTaskPrac* we must decide on the number of samples K to use for computing σ_e^2 with the WUNN (Algorithm 17, line 17); and 4) we must decide on appropriate priors for the weights of the WUNN μ_0 and σ_0^2 (Algorithm 18, line 1).

Note that when updating the FFNN, we train with the full *MemoryBuffer* list over *TrainIter* iterations (Algorithm 18, line 32). We found that using the full memory buffer produced far more accurate cost-to-goal predictions over using a minibatch approach because the backpropagation algorithm can compute gradients over the entire training dataset, rather than only part of the training dataset, in each training iteration.

5.5 Experiments

In Section 5.4 we introduce a practical implementation of our framework to efficiently learn likely-admissible heuristics. In this section we run experiments on the 15-puzzle, 24-puzzle, 24-pancake and 15-blocksworld domains using this implementation. These are common planning domains, allowing for comparisons to be made with previous research results [Korf and Taylor, 1996, Felner and Adler, 2005, Helmert, 2010, Arfaee et al., 2011, Arfaee et al., 2010, Ernandes and Gori, 2004]. In all domains we use parameter values as per Table 5.1.

Name	Value
ϵ	1
NumTasksPerIter	10
NumTasksPerIterThresh	6
α_0	0.99
Δ	0.05
β_0	0.05
κ	0.64
MemoryBufferMaxRecords	$25,000$
μ_0	0
σ_0^2	10
q	0.95
K	100

Table 5.1: Parameter values used in 15-puzzle, 24-puzzle, 24-pancake and 15-blocksworld domain experiments.

The number of iterations *NumIter* depends on the experiment as we describe further below. We set η as a function of *NumIter* by specifying $\beta_{NumIter} = 0.00001$ and solving $\eta^{NumIter}\beta_0 = \beta_{NumIter}$. Since β controls the strength of the prior, setting $\beta_{NumIter}$ small ensures that the importance of the prior relative to the likelihood becomes negligible by the end of training regardless of the assignment for *NumIter*.

To reduce variance when training the WUNN we use the local reparameterisation trick that samples from the neuron distribution implied from the weight connections to that neuron, rather than from the weight distributions directly. [Kingma et al., 2015]. All the networks use *relu* activations, have a single hidden layer and are trained with the Adam optimiser. For the WUNN we use an initial learning rate of 0.01 and $S = 5$ Monte Carlo samples to approximate the objective function in equation 5.4. For the FFNN we use an initial learning rate of 0.001. The network weights are initialised with He Normal initialization [He et al., 2015] while the biases are initialised to 0. We use IDA* for planning [Korf, 1985]. When running the final learned heuristic on test tasks we use $\sigma_t^2 = \sigma_a^2$ - this makes the assumption that epistemic uncertainty is negligible for all states by the end of training.

For training the FFNN we train for *TrainIter* $= 1000$ iterations using the entire memory buffer at each iteration. For training the WUNN we train for *MaxTrainIter* $= 5000$ iterations using a minibatch size of *MiniBatchSize* $= 100$, and we use a sampling scheme that gives more weight to states with epistemic uncertainty greater than or equal to $\kappa\epsilon$. This can be achieved by sampling from the following (unnormalised) distribution, without replacement:

$$f(x) = \begin{cases} \exp(\sigma_e(x)) & \text{if } \sigma_e^2(x) \geq \kappa\epsilon, \\ \exp(-C) & \text{otherwise,} \end{cases}$$

where $C > 0$. Since $\sigma_e \geq 0$ this distribution gives more weight to states with epistemic uncertainty that still need to be reduced below the required threshold. We use $C = 1$ in our implementation. In all our experiments we report average statistics over multiple runs and we include detailed tables with standard deviations for each statistic in Appendix Section C.2.

5.5.1 Suboptimality Experiments for the 15-puzzle

For the 15-puzzle we use a low-level feature representation of the raw game state and 20 hidden neurons for both the FFNN and WUNN. Previously, a representation was described that requires 16^2 bits where for each 16 puzzle number, the location of that number is encoded with a 16 bit one-hot vector [Ernandes and Gori, 2004]. We use a more efficient encoding that requires only $16 \times 2 \times 4$ bits where for each 16 puzzle number the horizontal and vertical locations of that number are encoded with a 4 bit one-hot vector.

For the first experiment we aim to show that our approach results in low suboptimality. We run the Algorithm *LearnHeuristicPrac* to learn the neural heuristic. We run the procedure for $NumIter = 50$ iterations, $t_{max} = 60$ and $MaxSteps = 1,000$. Once training is completed, we test the performance of our algorithm on the standard 100 15-puzzle benchmark tasks [Korf, 1985] that have an average optimal cost-to-goal of 53.05.

One problem that arises when using a representation of the raw game state is that such a representation does not exhibit aleatoric uncertainty since the network can, in principle, learn a deterministic output for each input state. To alleviate this, we use dropout which is a popular technique to inject noise into the training process by randomly dropping out neurons in the hidden layer of the network during training. Dropout is commonly seen as a way to reduce overfitting [Srivastava et al., 2014] but can also be viewed as a technique to estimate aleatoric uncertainty [Osband, 2016]. In this context, after dropout is applied, the latent feature-space of the hidden layer is insufficient to deterministically explain the cost-to-goal observations which induces aleatoric uncertainty. In our setup, we use a dropout rate of 2.5% when training the FFNN. This dropout rate was chosen by trial-and-error. From out tests, we found that the dropout rate played an important role in experimental results. If the dropout rate is set too small, then not enough noise is injected into the process network resulting in negligible aleatoric uncertainty. If the dropout rate

is set too large, then too much noise is injected into the training process resulting in exaggerated aleatoric uncertainty.

In addition, we train a second neural heuristic that uses *LearnHeuristicPrac* but we make a modification that the FFNN has one output neuron and therefore planning and learning is done with $\hat{y}$ only. This is consistent with previous approaches that learn neural heuristics [Arfaee et al., 2010, Arfaee et al., 2011, Ernandes and Gori, 2004]. Results are shown in Table 5.2 and are averages over 10 independent runs. The experiment that uses a single output FFNN has N/A under the α column while MD corresponds to using the admissible Manhattan Distance heuristic. Columns 'Generated', 'Time', 'Subopt' and 'Optimal' are the average nodes generated, planning time (in seconds), suboptimality [6] and percentage tasks solved optimally respectively.

We see that the single output FFNN produces a neural heuristic with a high suboptimality of 10.8%. This result conforms to what was previously described in the literature in that errors compound with each iteration [Arfaee et al., 2010, Arfaee et al., 2011]. Using our approach, we can achieve far lower suboptimality - for example, only 2.5% for $\alpha = 0.90$. However, we note that using $\alpha = 0.9$ expands 9.76 times more nodes. This suggests that the improved optimality achieved in our algorithm comes at the price of increased planning time. However, we can adjust the trade-off between planning time and suboptimality by varying α. We see that as α increases the trend is for suboptimality to decrease while nodes generated increases.

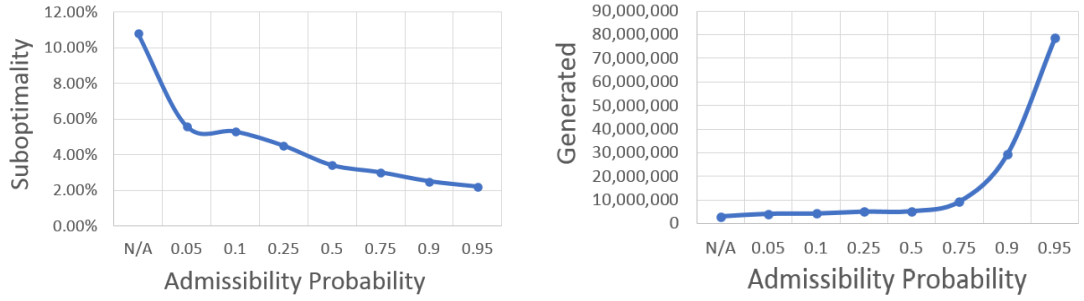
α	Time	Generated	Subopt	Optimal
MD	80.7	363,028,080	0.0%	100%
0.95	74.6	78,787,262	2.2%	67.8%
0.9	**26.7**	**29,342,747**	**2.5%**	**65.2%**
0.75	8.7	9,357,055	3.0%	59.0%
0.5	5.1	5,284,645	3.4%	52.3%
0.25	4.9	5,107,840	4.5%	38.3%
0.1	3.9	4,285,483	5.3%	30.7%
0.05	3.8	4,189,753	5.6%	25.3%
N/A	**2.3**	**3,071,956**	**10.8%**	**10.9%**

Table 5.2: Suboptimality results for 15-puzzle. N/A corresponds to the single output FFNN; MD corresponds to the Manhattan Distance heuristic.

Figure 5.5 plots suboptimality and nodes generated as a function of α. We see from the figure that the trade-off between suboptimality and nodes generated is non-

[6] Average suboptimality is computed by taking $u_i = \frac{y_i}{y_i^*} - 1$ for each test task where y_i is the cost-to-goal obtained from the planner and y_i^* is the optimal cost-to-goal, then averaging all u_i values.

linear. In fact, while suboptimality decreases linearly, nodes generated increases exponentially. This exponential increase in the nodes generated would be an important consideration in the setting of α for any practical application that uses our proposed framework.



(a) (b)

Figure 5.5: Suboptimality and nodes generated for the 15-puzzle admissibly experiments. N/A corresponds to the single output FFNN.

We note that previous work that trained on optimal plans and that used 4 separate neural networks to get better heuristic estimates was able to solve 63.86% tasks optimally [Ernandes and Gori, 2004]. Meanwhile our approach with $\alpha = 0.9$ is able to achieve a slightly higher percentage of 65.2%. [7] We use only a single network for planning while we do not have access to *any training data of optimal plans*.

5.5.2 Efficiency Experiments for the 15-puzzle

For the second experiment on the 15-puzzle we show that our approach to generating tasks using *GenerateTaskPrac* is more efficient than random task generation. In particular, previous work has suggested an alternative way to generate tasks of incremental difficulty [Arfaee et al., 2010, Arfaee et al., 2011]. The idea is to simply increment the number of steps taken back from the goal at each iteration using a parameter *LengthInc*. For example, if *LengthInc* = 10 we would first generate *NumTasksPerIter* tasks by taking 10 steps at random from the goal state, then 20 steps, then 30 steps and so on.

Two key considerations are required to illustrate efficiency. Firstly, an efficient algorithm would learn about the domain from a small number of tasks because it would generate tasks in accordance with the amount of new knowledge it can gain from solving them. Secondly, an efficient algorithm would solve new tasks quickly because it would generate tasks that are incrementally more difficult than previously solved tasks.

[7] The statistic 63.86% was obtained from 700 test tasks with an average optimal cost-to-goal of 52.62 that were not made public. Meanwhile, we test on the standard 100 15-puzzle benchmark tasks that have an average optimal cost-to-goal of 53.05.

Given these considerations our experimental setup is as follows: we run *LearnHeuristicPrac* but we train a FFNN with only a single output neuron that plans and learns with $\hat{y}$. For this experiment we keep the number of tasks small, so $NumIter = 20$ and we keep the time limit to solve each task small as well, so $t_{max} = 1$. We then train additional FFNN networks using *LearnHeuristicPrac* but this time instead of using *GenerateTaskPrac* to generate tasks, we generate tasks using fixed steps back from the goal with $LengthInc \in \{1, 2, 4, 6, 8, 10\}$.

Once the neural heuristics are trained we test performance as follows: we generate a set of 100 test tasks $\{\mathcal{T}_i^{test}\}_{i=1}^{100}$. Task $\mathcal{T}_k^{test}$ is generated by taking k steps back at random from the goal, so that each task in the test set should be more difficult to solve than the previous one. We then give each of the neural heuristics a time limit of 60 seconds to solve as many test tasks as possible. We repeat this process independently 10 times across each neural heuristic and we furthermore run the experiment over 10 independent runs.

Table 5.3 shows our results. The experiment that uses *GenerateTaskPrac* to generate tasks has GTP under the *LengthInc* column. Columns 'Solved Train' and 'Solved Test' are the average percentage of tasks solved during training and at test time respectively. Consider the case $LengthInc = 1$. In this case 100% of training tasks are solved. However, because each successive iteration only takes 1 step back from the goal and there are only a small number of tasks to learn from, this neural heuristic only solves 38.59% of the test tasks. Now consider the case $LengthInc = 10$. In this case large steps are taken back from the goal but because there is a short time limit to solve each task, this neural heuristic can only solve 11.80% of training tasks. As a result, the network doesn't learn from enough tasks during training and can therefore only solve 31.83% of the test tasks.

Our approach finds a balance between these two restrictions. The training tasks generated are incrementally more difficult than previously observed training tasks because we stop execution of the algorithm as soon as we observe a state with high epistemic uncertainty. We can therefore solve 93.3% of tasks during training. Furthermore, because our approach seeks uncertainty when generating tasks, each new task that is trained on provides useful domain knowledge for the neural heuristic to learn from. Therefore, we can solve 60.59% of the test tasks. In fact, our approach outperforms all the values of $LengthInc \in \{1, 2, 4, 6, 8, 10\}$ in terms of performance on the test tasks. A further advantage of our approach is that the parameter *LengthInc*, which is domain dependant, does not need to be tuned for each new domain - we use the same β_0 and the same approach to set η for all experiments across the domains.

LengthInc	Solved Train	Solved Test
1	**100%**	**38.6%**
2	95.1%	48.2%
4	61.8%	51.4%
6	36.2%	39.6%
8	19.2%	35.2%
10	**11.8%**	**31.8%**
GTP	**93.3%**	**60.6%**

Table 5.3: Efficiency results for 15-puzzle. GTP corresponds to using *Generate-TaskPrac*.

5.5.3 Experiments for Other Domains

We run the same experiments for suboptimality on the 24-puzzle, 24-pancake and 15-blocksworld domains. These domains have large state-spaces and so using a low-level representation is more difficult than in the case of the 15-puzzle. Instead, we use high-level feature representations as input to the network that are similar to previous approaches [Arfaee et al., 2010, Arfaee et al., 2011]. The features used make extensive use of pattern databases (PDBs) [Korf and Felner, 2002]. PDBs are a general method of generating admissible heuristics, making them a good choice of features for a variety of domains.

We use $NumIter = 75$, $t_{max} = 5 \times 60$ and $MaxSteps = 5000$ for these domains due to their increased complexity over the 15-puzzle. We use 8 neurons for the hidden layer for both the FFNN and WUNN and we do not use dropout because the feature representations now naturally induce aleatoric uncertainty.

As discussed in Section 5.2.4, it is possible to take advantage of known admissible heuristics when planning under a bootstrap learning approach. In this section, this is achieved by using a heuristic that is the maximum of the model prediction and all the admissible heuristics passed as features to the network, as further discussed below for each domain.

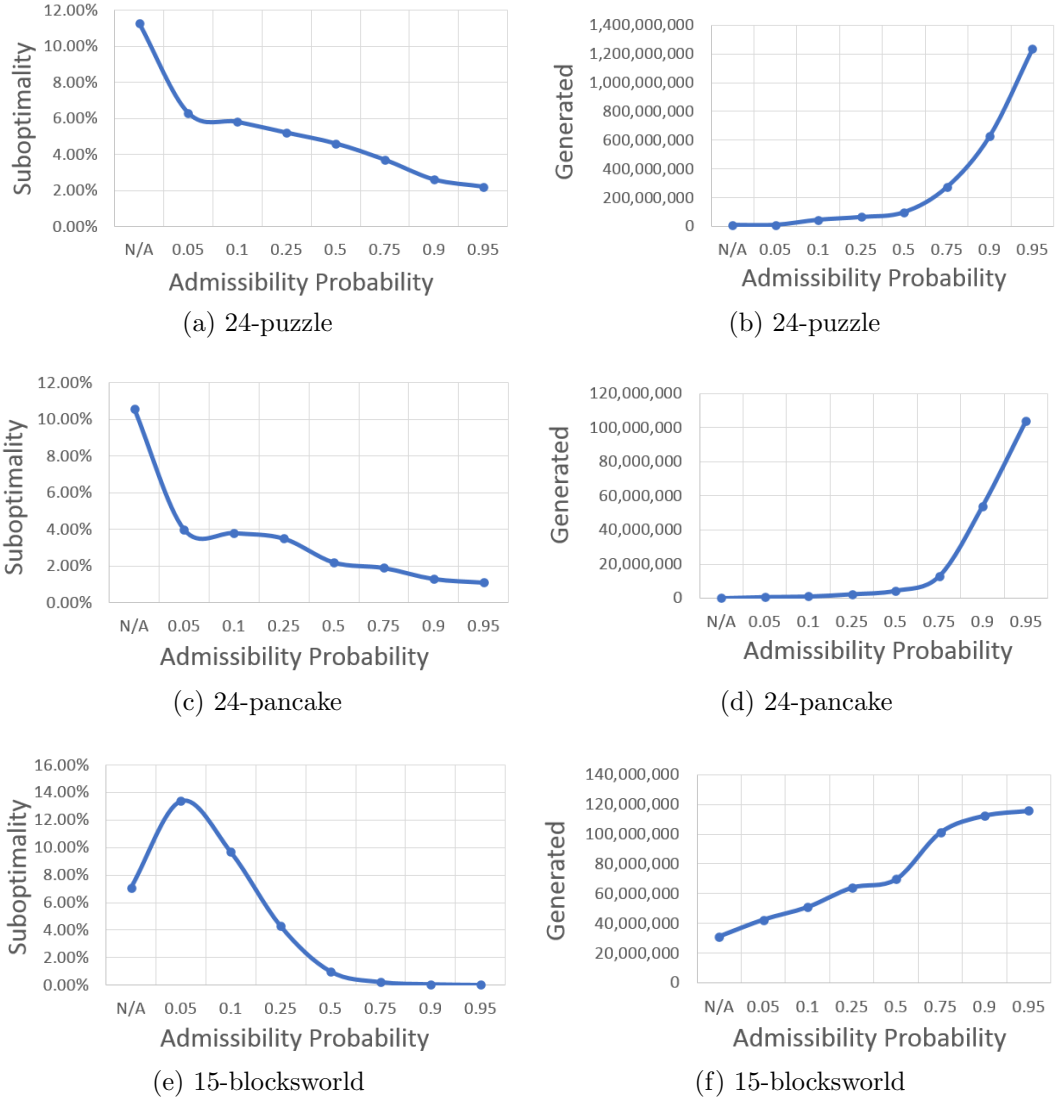
24-puzzle: we use 2 sets of disjoint 5-5-5-5-4 PDBs and their sums as features (f1-f12). Additionally, we have a feature with the number of out-of-place tiles (f13), a feature for the Manhattan distance (f14) and a feature for the position of the blank tile (f15). The total number of features for this domain is 15 comprising of 14 admissible heuristics (f1-14). For testing we use the standard 50 24-puzzle benchmark tasks that have an average optimal cost-to-goal of 100.78 [Korf and Taylor, 1996].

24-pancake: we use 2 sets of location-based disjoint 5-5-5-5-4 PDBs [Yang et al.,

2008] and their sums as features (f1-f12). Additionally, we have a binary feature indicating whether the middle pancake is out of place, and the number of the largest out-of-place pancake. The total number of features for this domain is 14 comprising of 12 admissible heuristics (f1-f12). For testing, we generate 50 random tasks that have an average optimal cost-to-goal of 22.56.

15-blocksworld: we use 12 4-block PDBs (f1-f12), the number of out of place blocks (f13), and the number of stacks of blocks (f14) as features. The total number of features for this domain is 14 comprising of 13 admissible heuristics (f1-f13). The goal state is chosen as the state with all blocks in one stack. For testing, we generate 50 random tasks that have an average optimal cost-to-goal of 21.72.

We provide more details on the PDBs used for each domain in Appendix Section C.1.



(a) 24-puzzle (b) 24-puzzle

(c) 24-pancake (d) 24-pancake

(e) 15-blocksworld (f) 15-blocksworld

Figure 5.6: Suboptimality and nodes generated for the 24-puzzle, 24-pancake and 15-blocksworld experiments. N/A corresponds to the single output FFNN.

α	Time	Generated	Subopt	Optimal
		24-puzzle		
PDB	–	65,135,068,005	0.0%	100%
0.95	2,665	1,233,965,823	2.2%	28.8%
0.9	**1,371**	**628,101,474**	**2.6%**	**20.0%**
0.75	549	274,003,465	3.7%	5.2%
0.5	189	99,244,234	4.6%	1.2%
0.25	121	66,147,586	5.2%	0.0%
0.1	87	46,988,530	5.8%	0.0%
0.05	84	40,046,361	6.3%	0.0%
N/A	25	11,719,659	11.3%	0.0%
Boot	**274**	**164,589,698**	**6.1%**	–
		24-pancake		
Gap	0.03	48,599.5	0.0%	100%
0.95	365	104,132,601	1.1%	76.0%
0.9	**199**	**54,089,822**	**1.3%**	**72.4%**
0.75	54	13,001,211	1.9%	59.2%
0.5	20	4,530,281	2.2%	53.2%
0.25	12	2,511,066	3.5%	37.2%
0.1	8	1,162,755	3.8%	30.8%
0.05	5	871,908	4.0%	30.8%
N/A	1	210,622	10.6%	8.4%
Boot	**2**	**1,856,645**	**10.3%**	–
		15-blocksworld		
0.95	56	115,691,681	0.02%	99.6%
0.9	**54**	**112,390,208**	**0.07%**	**98.4%**
0.75	51	101,109,757	0.23%	95.6%
0.5	38	69,663,441	1.0%	84.8%
0.25	44	63,963,572	4.3%	50.8%
0.1	36	50,951,658	9.7%	34.0%
0.05	29	42,499,655	13.4%	24.0%
N/A	21	31,178,090	7.1%	38.4%
Boot	**16**	**3,651,438**	**6.9%**	–

Table 5.4: Suboptimality results for 24-puzzle, 24-pancake and 15-blocksworld. 'Boot' corresponds to the original bootstrap Algorithm [Arfaee et al., 2010]; 'Gap' corresponds to the gap heuristic [Helmert, 2010]; 'PDB' corresponds to a disjoint 6-6-6-6 PDBs with partially created disjoined 8-8-8 PDBs heuristic [Felner and Adler, 2005].

Domain	plan with $\hat{y}$	plan with y^{α}
15-puzzle	1.32	2.67
24-puzzle	6.03	20.52
24-pancake	2.34	15.85
15-blocksworld	4.38	6.54

Table 5.5: Training runtime in hours.

Table 5.4 show our results which average over 5 independent runs. The rows with 'Boot' under the α column are results from the original bootstrap learning Algorithm [Arfaee et al., 2010] as discussed in Section 5.2.4. The row with 'Gap' under the α column is the domain specific gap heuristic for the 24-pancake [Helmert, 2010] and 'PDB' is a memory intensive heuristic that comprises of disjoint 6-6-6-6 PDBs with partially created disjoined 8-8-8 PDBs for the 24-puzzle [Felner and Adler, 2005].[8]

We see that, as in the case of the 15-puzzle, the single output FFNNs have high suboptimality. With our approach we can achieve far lower suboptimality, albeit at the price of longer planning times. For example, with $\alpha = 0.9$ we obtain 2.6%, 1.3% and 0.07% suboptimality while previous approaches obtain 6.1%, 10.3% and 6.9% suboptimality for the 24-puzzle, 24-pancake and 15-blocksworld domains respectively.[9]

As in the case of the 15-puzzle, the trend is that as α increases, suboptimality decreases while nodes generated increases. These experiments also illustrate that our implementation is robust as we use the same parameters across three domains and obtain similar outcomes. We also emphasise that our implementation works with both low-level and high-level feature representations, as we use a low-level feature representation for the 15-puzzle domain and high-level feature representations for the 24-puzzle, 24-pancake and 15-blocksworld domains.

A final point of discussion is that our approach to learning heuristics takes longer to train because solving tasks with a likely-admissible heuristic requires more nodes to be generated during planning. Runtimes for the different domains are shown in Table 5.5.

[8]For clarity, the 'Boot' and 'PDB' results are taken directly from their respective papers.

[9]For testing, the 'Boot' results are derived from the standard 50 benchmark tasks for the 24-puzzle, 1000 tasks with an average optimal cost-to-goal of 22.75 for the 24-pancake and 200 tasks with an average optimal cost-to-goal of 22.73 for the 15-blocksworld.

5.6 Future Work

The work presented in this chapter has demonstrated the validity of a bootstrap learning approach for efficient learning of likely-admissible heuristics. As the focus of this chapter is on the learning procedure, we have restricted our attention to the simpler classical planning setting. We have furthermore used planning domains with unique goal states. An important area of future research is therefore the extension of the methodologies described in this chapter to more complex planning settings. These include:

- planning domains with multiple goal states;
- lifted planning domains;[10] and
- the general MDP setting.

All these extensions are conceptually straightforward. The extension to multiple goal states can be incorporated into our proposed framework as follows:

- When running *GenerateTask*, sample a goal state s_g from all possible goal states in $\mathcal{S}_g$ and then proceed to generated a task with this goal state.
- The feature function F must be extended to encode a representation of the goal state in addition to the current state i.e. $x = F(s, s_g)$.
- The model $\mathfrak{M}$ must be extended to accept as input the expanded feature vector x.

Meanwhile, accommodating lifted domains requires a judicious choice for the feature function F and model $\mathfrak{M}$. For example, previous work has shown that by using a particular choice of representation together with a convolutional neural network model, a heuristic function can be learned that generalises across the Sokoban domain [Groshev et al., 2017]. Then as long as this neural network architecture can be augmented with a mechanism to effectively model epistemic and aleatoric uncertainty, we can incorporate it into our proposed framework. Finally, previous work has extended heuristic-based planning to the general MDP setting [Hansen and Zilberstein, 1998, Hansen and Zilberstein, 2001, Dai et al., 2011] and the algorithms presented in this chapter should be directly applicable to this setting as well.

While conceptually straightforward, each of these extensions may uncover practical challenges that would need addressing. For example, extending to multiple goal

[10] An example of a lifted domain is the n-puzzle domain that instantiates tasks from any valid n-puzzle tasks i.e. 8-puzzle, 15-puzzle, 24-puzzle etc. A further example is the Sokoban domain where the size of the maze as well as the locations of walls, boxes and storage locations can change in each task.

states if the size of $\mathcal{S}_g$ is large will require important considerations such as 1) how to encode the goal states so that a function approximator can learn to generalise from as few samples as possible; and 2) a sampling scheme that samples goal states efficiently, as naïve uniform sampling may prove too slow.

A tangential, but equally important, direction for future research is the implementation and comparison between different instances of the conceptual framework presented in Section 5.3. In this chapter, we used a fairly simple Bayesian neural network based on WUNNs that uses independent Gaussian weights and a Gaussian response. Clearly, a Gaussian response is not ideal to model a heuristic function since the cost-to-goal is known to be non-negative. A better choice of output distribution may produce stronger predictive cost-to-goal performance as well as more accurate measures of aleatoric uncertainty. Meanwhile, alternative methods to estimate epistemic uncertainty in Bayesian neural networks exist [Gal and Ghahramani, 2016, Hernandez-Lobato and Adams, 2015, Sun et al., 2017]. For example, previous work has developed a model that learns structured weights by taking into account weights correlations [Sun et al., 2017]. Such a method may provide better estimates of epistemic uncertainty that improve efficiency.

Finally, in Appendix Section C.2 we present the same set of tables for the experiments of Section 5.5 but that also include standard deviation statistics. As can be seen from these tables, the implementation used in this chapter tends to have extremely high variance. For example, as shown in Table C.3, with $\alpha = 0.9$ we generate $628,101,474$ nodes over 5 independent runs for the 24-puzzle domain. The standard deviation over these 5 runs is $539,128,139$ which is extremely high. Therefore, an important area for future research is to develop variance reduction techniques for learning heuristics by bootstrapping.

5.7 Conclusion

This chapter has introduced a conceptual framework to efficiently learn likely-admissible heuristics for a domain. Our approach learns via bootstrapping and so has the advantage of not requiring an existing dataset of plans for training.

In terms of the complexity metrics discussed in Section 2.3.1, utilising a strong heuristic can greatly reduce computation complexity when planning. However, such a heuristic need to be learned. In this chapter we have introduced a framework to efficiently learn such heuristics.

Our proposed framework uses epistemic uncertainty to efficiently explore tasks-space and generate training tasks that are of the right level to learn from based on the

knowledge encoded in the current heuristic. This is more sample efficient than random task generation used in previous bootstrap learning approaches, thus requiring less training tasks to learn from.

Furthermore, previous approaches to bootstrap learning of heuristics have a failing where errors compound with each iteration of the learning procedure, resulting in a final heuristic that produces high suboptimality. In this chapter we have used epistemic and aleatoric uncertainty to mitigate this issue by planning with a likely-admissible heuristic that produces likely-optimal plans for training.

The conceptual framework introduced in this chapter can operate with a broad range of models that produce measures of epistemic and aleatoric uncertainty. In order to evaluate the validity of our framework, we instantiate a particular model based on Bayesian neural networks and also provide algorithms that overcome the practical issues in implementing the conceptual framework.

We demonstrate our practical algorithm empirically on four domains: the 15-puzzle, 24-puzzle, 24-pancake and 15-blocksworld domains. In our experiments we show that our approach to learning heuristics is more sample efficient than competing methods, while producing a final heuristic with low suboptimality metrics that outperform competing methods.

Chapter 6

Conclusion

The Reinforcement Learning (RL) paradigm is highly appealing and can, in principle, be used to solve a diverse range of important tasks in the field of Artificial Intelligence (AI). Unfortunately, standard RL algorithms tend to be highly sample inefficient, and so the framework is hampered by an inability to scale to large, real-world tasks. As discussed in Section 2.2 some of the major challenges to overcome this sample inefficiency include: sparse rewards, the 'curse of dimensionality' and non-transferable representations. In this thesis we introduce three independent frameworks that improve sample efficiency in RL by leveraging prior knowledge to address these challenges.

- In Chapter 3 we introduce a reward shaping framework called Belief Reward Shaping (BRS) that can be used in conjunction with any model-free RL algorithm, such as Q-learning. BRS differs from previously introduced potential-based reward shaping frameworks in that it imposes a prior on the environment reward distribution directly. This prior can encode knowledge that biases the agent to achieve certain sub-tasks within a task. Provided this prior is well-specified, it can then be used to improve sample efficiency by making the total reward that the agent receives less sparse. The advantage of BRS is that it can encode prior knowledge using the full transition information directly, while still having the policy invariance property of potential-based reward shaping frameworks.
- In Chapter 4 we introduce two object-oriented frameworks that are used in conjunction with the model-based KWIK-R_{max} algorithm. The first framework introduces Deictic OO-MDPs that uses deictic predicates to compactly represent the transition dynamics for certain classes of domains. The second framework extends previous work on Propositional OO-MDPs to compactly represent the reward dynamics for certain classes of domains. We then intro-

duce a KWIK-R_{max} based algorithm $DOORMAX_D$ to efficiently learn these dynamics. Since our frameworks learn the dynamics for an entire domain, they can be learned on some simpler source tasks of a domain and then zero-shot transferred to more complex target task of the domain. This improves sample efficiency as the agent does not have to waste samples relearning the dynamics for each new task it encounters.

- Naïve planning algorithms, like value iteration, struggle to scale as the size of the state-space or action-space grows because of the 'curse of dimensionality'. An alternative approach is to utilise heuristic-based planners. Such heuristic-based planners can greatly improve planning performance, but require prior knowledge in the form of a strong heuristic in order to be effective. In Chapter 5 we introduce a bootstrapping framework to efficiently learn likely-admissibly heuristics for planning domains. Our approach leverages epistemic uncertainty to efficiently explore task-space by generating tasks that have start states with high epistemic uncertainty to learn from. Furthermore, previous bootstrap approaches have been shown to produce high suboptimality because errors compound at each iteration of the bootstrap procedure. In our framework, we leverage epistemic and aleatoric uncertainty to ensure that the heuristic learns from likely-optimal plans during training. Once learned, the heuristic can then be transferred to new tasks of the domain to produce likely-optimal plans with low suboptimality.

In the remainder of this concluding chapter we discuss future work in Section 6.1 and close with final remarks in Section 6.2.

6.1 Future Work

In Sections 3.5, 4.8 and 5.6 we discuss future work of Chapters 3, 4 and 5 respectively. In this section, we take a step back from the individual chapters and consider directions for future work based on a holistic view of these chapters with respect to the central themes of this thesis.

6.1.1 General Representations

In Chapter 4 we use object-oriented representations to compactly represent the transition and reward dynamics for certain classes of domains. In Section 3.5 of Chapter 3 we discuss that reward shaping techniques have relied on abstract state-space representations to accommodate transfer. Finally, in Chapter 5 we make use of neural network models to learn a heuristic function that transfers across a domain.

Neural networks implicitly learn a latent representation that is stored in their hidden layers.

Each of these techniques, therefore, uses a different representation that is suited to its application. A major outstanding question in the field of RL is whether there exist more general representations, and if so, what are these and how can they be learned?

To make the notion of a general representation concrete, consider Chapter 4 where we use logical statements over object classes to compactly represent the transition and reward dynamics for the Sokoban domain. Note, however, that the state-space is fully grounded and we learn a value function (or policy) over this grounded state-space when planning.

Previous work has introduced an object-oriented approach that learns generalised value functions by approximating the value function of a grounded state with a sum of class-level value functions [Guestrin et al., 2003, Guestrin, 2003]. This approach works for the domain used in that research - a simplified strategy game invented by the authors called *Freecraft*. However, it is not suitable for a domain like Sokoban. This is because, in Sokoban, a small change in a state can have a large impact on the value of that state that a class-level approximation cannot easily capture. To see why, suppose you are given some high-level information about a Sokoban task such as: nine out of ten boxes are at storage locations. Unfortunately, this information is not sufficient to determine that this is a high-value state because, for example, the grounded state could be a deadlock state from which the task cannot be solved.

In totality, we observe that the object-oriented representations presented in this thesis can compactly represent the dynamics of Sokoban, but not the value function. Meanwhile, there may exist domains where the value function is well suited to some form of object-oriented representation, while the dynamics are not. A general representation then, is one that can operate seamlessly across a large class of domains, and within all components of these domains. We see from this example that the object-oriented representations described in this thesis may not be general enough.

Possibly, the most fruitful direction with current known methods to learn such general representations is to leverage Deep Learning techniques that learn disentangled representations from low-level data [Higgins et al., 2017]. Such representations have been shown to produce robust factors with good generalisation properties. Another interesting idea is the notion of inheritance inspired by object-oriented programming. For example, most strategy games have an abstract 'builder' class whose job is to build and repair structures. While the visual characteristics and nuances of the class differ from game to game, it may be possible to learn a representation about

the abstract class that generalises to game-specific classes that inherit from it.

6.1.2 Abstract Transfer

In Chapters 4 and 5 we have introduced frameworks showing that transfer is possible across tasks of a domain. In this thesis, we have loosely defined a domain as a set of 'related' tasks. For example, the Sokoban domain comprises all Sokoban tasks. In this case, transfer in the domain implies transfer between Sokoban tasks.

While this is a good first step, it is not comparable with human-level transfer that is far more abstract. As a concrete example of such abstract transfer, consider a human player who plays strategy video games. Clearly, mastering one strategy game aids the human player when starting a new strategy game. This prior knowledge from the previous game applies even though the new strategy game may be quite different from the original and have, for example, different types of resources, units and structures available. Despite such major differences, humans are able to transfer much abstract knowledge from previous experiences to new tasks.

Therefore, an important area of future work is the development of techniques to learn more abstract forms of transfer that expand the notion of 'task relatedness'. How such abstract transfer might be achieved is a challenging but important research question in the field of RL and, more generally, AI.

One promising direction for this goal is the combination of object-oriented representations with Deep Learning models. Deep Learning models typically leverage neural network architectures with many hidden layers to learn abstract representations through their hierarchical structure. One would hope that such abstract representations would lead to abstract transfer properties. Unfortunately, these abstract representations have shown evidence of being rigid with respect to the tasks they are trained on, therefore making them unsuitable for transfer [Kansky et al., 2017]. However, recent work has used Deep Learning models in conjunction with object-oriented representations to learn object embeddings that have stronger abstract transfer properties [Woof and Chen, 2018].

6.1.3 Integration

In this thesis we have introduced three frameworks that leverage prior knowledge to improve the sample efficiency of RL. While the frameworks share this common thread, each is independent of the others. An important area of future research in RL is to develop techniques that take seemingly independent frameworks and integrate them to extract their strengths. Some examples based on this thesis:

- The object-oriented representations learned with $DOORMAX_D$. The $DOORMAX_D$ algorithm explores an environment by assuming that all unexplored transitions yield the highest possible reward. This reward is the same for all unexplored transitions. Therefore, the agent explores unknown transitions uniformly. Furthermore, the agent must explore all transitions under such a policy. Meanwhile, Reward Shaping frameworks, like Belief Reward Shaping, bias the agent to seek (or avoid) certain transitions by giving the agent a shaping reward when reaching these transitions.

  An integration between these two frameworks could then look at techniques that prioritise exploration in $DOORMAX_D$ (or any KWIK-R_{max} algorithm) based on shaping rewards so that more fruitful transitions, as determined by the shaping rewards, are explored first. Perhaps under such a prioritisation, a near-optimal policy can be learned that mitigates the need for the agent to explore the entire environment.

- In $DOORMAX_D$, each time the agent interacts with its environment and gains experience, an exact planning algorithm is called to learn an optimal policy under the current (incorrect) model of the dynamics. In our experiments we use value iteration. Meanwhile, our framework to efficiently learn likely-admissibly heuristics applies to planning domains with known dynamics. Therefore, with the current setup, only once the environment dynamics are completely learned for a domain with $DOORMAX_D$ is it possible to switch to our heuristic-based approach for efficient planning.

  However, planning with the incorrect models in $DOORMAX_D$ still relies on a non-heuristic based planning algorithm like value iteration, which can be inefficient. The challenge with using a heuristic-based approach in this context is that the form of a heuristic function is tied explicitly to the dynamics, and in $DOORMAX_D$ the dynamics used for planning are constantly being updated until the models are fully learned.

  One potential way around this, as discussed in Section 4.8, is to utilise Curriculum Learning strategies to ensure that the tasks used when learning the dynamics are not too complex. However, for some domains even simpler tasks may be too complex to solve with an inefficient planning algorithm.

  An alternative direction is to develop more efficient planning algorithms that can integrate with a KWIK-R_{max} based algorithm like $DOORMAX_D$. Previous work, for example, has introduced more efficient forms of value iteration that eliminate redundant Bellman backups [Wingate, 2004, Wingate and Seppi, 2004]. While such enhancements are extremely valuable, it is unlikely that they

can ever outperform a planner that utilises prior knowledge in the form of a strong heuristic. Therefore, it would be beneficial to assess whether a heuristic-based approach similar to that introduced in Chapter 5 can be utilised with algorithms like $DOORMAX_D$, where the dynamics used for planning are being updated as the agent learns.

6.2 Final Remarks

A longstanding goal in the field of AI is the construction of autonomous agents that operate effectively in the real-world. While recent achievements in the field of AI have been successful at solving an impressive array of complex tasks, many of these successes have relied on high computational resources and are highly sample inefficient.

While there certainly are merits in using the limits of available computational resources to solve such tasks, it makes intuitive sense that a ceiling will be reached with sample inefficient algorithms. In comparison to such algorithms, humans have a remarkable ability to solve new tasks in a sample efficient way. This is because they leverage prior knowledge from past experiences and transfer it effectively. It is beneficial to use human cognition as inspiration when developing autonomous agents. By this measure, the key to improving the sample efficiency of autonomous agents is to increase their capability to leverage prior knowledge effectively.

This thesis takes a step in that direction with a focus on RL, a subfield of AI. In particular, we have introduced three novel frameworks that incorporate prior knowledge through reward shaping (Chapter 3), object-oriented representations (Chapter 4) and bootstrap learning of heuristics (Chapter 5). Each of these frameworks has been shown to improve sample efficiency in RL.

The frameworks introduced in this thesis show encouraging results. However, there is much scope for improvement with future work. In particular, we highlight the need for improved general representations, abstract transfer and integration in RL. Our hope is that with an increased focus in this research direction, new techniques can build on the ideas of this thesis to achieve the longstanding goals in AI.

Bibliography

[Abel et al., 2018] Abel, D., Arumugam, D., Lehnert, L., and Littman, M. (2018). State abstractions for lifelong reinforcement learning. *Proceedings of the 35th International Conference on Machine Learning*, pages 10–19.

[Arfaee et al., 2010] Arfaee, S. J., Zilles, S., and Holte, R. C. (2010). Bootstrap learning of heuristic functions. *Proceedings of the 3rd Annual Symposium on Combinatorial Search*, pages 52–60.

[Arfaee et al., 2011] Arfaee, S. J., Zilles, S., and Holte, R. C. (2011). Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2089.

[Bertsekas, 1987] Bertsekas, D. P. (1987). *Prentice-Hall*. MIT Press.

[Blundell et al., 2015] Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural networks. *Proceedings of the 25th International Conference on Machine Learning*.

[Boutilier et al., 2001] Boutilier, C., Reiter, R., and Price, B. (2001). Symbolic dynamic programming for first-order MDPs. *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 690–697.

[Brafman and Tennenholtz, 2002] Brafman, R. I. and Tennenholtz, M. (2002). R-MAX - a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231.

[Clark and Amodei, 2016] Clark, J. and Amodei, D. (2016). Faulty reward functions in the wild. https://openai.com/blog/faulty-reward-functions.

[Dai et al., 2011] Dai, P., Weld, D. S., and Goldsmith, J. (2011). Topological value iteration algorithms. *Artificial Intelligence*, 42(1):181–209.

[Devlin and Kudenko, 2012] Devlin, S. and Kudenko, D. (2012). Dynamic potential-based reward shaping. *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, pages 433–440.

[Dietterich, 1998] Dietterich, T. (1998). The MAXQ method for hierarchical reinforcement learning. *Proceedings of the 15th International Conference on Machine Learning*.

[Dinh et al., 2007] Dinh, H. T., Russell, A., and Su, Y. (2007). On the value of good advice: The complexity of A* search with accurate heuristics. *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pages 1140–1145.

[Diuk, 2010] Diuk, C. (2010). *An Object-Oriented Representation for Efficient Reinforcement Learning*. PhD thesis, Rutgers The State University of New Jersey-New Brunswick.

[Diuk et al., 2008] Diuk, C., Cohen, A., and Littman, M. L. (2008). An object-oriented representation for efficient reinforcement learning. *Proceedings of the 25th International Conference on Machine learning*, pages 240–247.

[Dubey et al., 2018] Dubey, R., Agrawal, P., Pathak, D., Griffiths, T. L., and Efros, A. A. (2018). Investigating human priors for playing video games. *Thirty-fifth International Conference on Machine Learning*.

[Džeroski et al., 2001] Džeroski, S., DeRaedt, L., and Driessens, K. (2001). Relational reinforcement learning. *Machine Learning Journal*, 43:7–52.

[Ernandes and Gori, 2004] Ernandes, M. and Gori, M. (2004). Likely-admissible and sub-symbolic heuristics. *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 613–617.

[Felner and Adler, 2005] Felner, A. and Adler, A. (2005). Solving the 24 puzzle with instance dependent pattern databases. *Proceedings of the 6th International Symposium on Abstraction, Reformulation and Approximation*, pages 248–260.

[Gaina et al., 2019] Gaina, R. D., Lucas, S. M., and Perez-Liebana, D. (2019). Tackling sparse rewards in real-time games with statistical forward planning methods. *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*.

[Gal and Ghahramani, 2016] Gal, Y. and Ghahramani, Z. (2016). Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. *Proceedings of the 33rd International Conference on Machine Learning*, pages 1050–1059.

[Gelman et al., 1995] Gelman, A., Carlin, J. B., Stern, H. S., and Rubin, D. B. (1995). *Bayesian Data Analysis*. Chapman and Hall.

[Groshev et al., 2017] Groshev, E., Tamar, A., Srivastava, S., and Abbeel, P. (2017). Learning generalized reactive policies using deep neural networks. *CoRR abs/1708.07280*.

[Grzes and Kudenko, 2008] Grzes, M. and Kudenko, D. (2008). Online learning of shaping rewards in reinforcement learning. *Proceedings of the 8th International Conference on Artificial Neural Networks*.

[Grzes and Kudenko, 2009] Grzes, M. and Kudenko, D. (2009). Theoretical and empirical analysis of reward shaping in reinforcement learning. *Proceedings of the IEEE International Conference on Machine Learning and Applications*, pages 337–344.

[Guestrin, 2003] Guestrin, C. (2003). *Planning Under Uncertainty in Complex Structured Environments*. PhD thesis, Stanford University.

[Guestrin et al., 2003] Guestrin, C., Koller, D., Gearhart, C., and Kanodia, N. (2003). Generalizing plans to new environments in relational MDPs. *Proceedings of the 18th Joint Conference on Artificial Intelligence*, pages 1003–1010.

[Hailu and Sommer, 1999] Hailu, G. and Sommer, G. (1999). On amount and quality of bias in reinforcement learning. *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 1491–1495.

[Hansen and Zilberstein, 1998] Hansen, E. and Zilberstein, S. (1998). Heuristic search in cyclic and/or graphs. *Proceedings of the 15th National Conference on Artificial Intelligence*, pages 412–418.

[Hansen and Zilberstein, 2001] Hansen, E. and Zilberstein, S. (2001). LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129:35–62.

[Hare, 2019] Hare, J. (2019). Dealing with sparse rewards in reinforcement learning. *CoRR abs/1910.09281*.

[Hart et al., 1968] Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

[Harutyunyan et al., 2015] Harutyunyan, A., Devlin, S., Vrancx, P., and Nowé, A. (2015). Expressing arbitrary reward functions as potential-based advice. *Proceedings of the 29th AAAI Conference on Artificial Intelligence*.

[Haslum and Scholz, 2003] Haslum, P. and Scholz, U. (2003). Domain knowledge in planning: Representation and use. *Workshop on PDDL at International Conference on Automated Planning and Scheduling*.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR abs/1502.01852*.

[Helmert, 2010] Helmert, M. (2010). Landmark heuristics for the pancake problem. *Proceedings of the 3rd International Symposium on Combinatorial Search*, pages 109–110.

[Hernandez-Lobato and Adams, 2015] Hernandez-Lobato, J. M. and Adams, R. (2015). Probabilistic backpropagation for scalable learning of Bayesian neural networks. *Proceedings of the 25th International Conference on Machine Learning*.

[Hershkowitz et al., 2015] Hershkowitz, D. E., MacGlashan, J., and Tellex, S. (2015). Learning propositional functions for planning and reinforcement learning. *2015 AAAI Fall Symposium Series*.

[Hester and Stone, 2009] Hester, T. and Stone, P. (2009). Generalized model learning for reinforcement learning in factored domains. *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems*.

[Higgins et al., 2017] Higgins, I., Matthey, L., Pal, A., Burgess, C., Glorot, X., Botvinick, M., Shakir, M., and Lerchner, A. (2017). beta-VAE: Learning basic visual concepts with a constrained variational framework. *Proceed of the 5th International Conference on Learning Representations*.

[Ho et al., 2015] Ho, M. K., Littman, M. L., Cushman, F., and Austerweil, J. L. (2015). Teaching with rewards and punishments: Reinforcement or communication? *Proceedings of the 37th Annual Conference of the Cognitive Science Society*.

[Holte, 2010] Holte, R. C. (2010). Common misconceptions concerning heuristic search. *Proceedings of the 3rd Annual Symposium on Combinatorial Search*, pages 46–51.

[Jaakkola et al., 1994] Jaakkola, T., Jordan, M. I., and Singh, S. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, pages 1185–1201.

[Jin et al., 2018] Jin, C., Allen-Zhu, Z., Bubeck, S., and Jordan, M. I. (2018). Is Q-learning provably efficient? *Thirty-second Conference on Neural Information Processing Systems*, pages 4868–4878.

[Junghanns and Schaeffer, 2001] Junghanns, A. and Schaeffer, J. (2001). Sokoban: Enhancing general single agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251.

[Juozapaitis et al., 2019] Juozapaitis, Z., Koul, A., Ferm, A., Erwig, M., and Doshi-Velez, F. (2019). Explainable reinforcement learning via reward decomposition. *Workshop on Explainable Artificial Intelligence at International Joint Conference on Artificial Intelligence*, pages 47–53.

[Kakade, 2003] Kakade, M. S. (2003). *On the Sample Complexity of Reinforcement Learning*. PhD thesis, Gatsby Computational Neuroscience Unit, University College London.

[Kansky et al., 2017] Kansky, K., Silver, T., Mèly, D. A., Eldawy, M., Làzaro-Gredilla, M., Lou, X., Dorfman, N., Sidor, S., Phoenix, S., and George, D. (2017). Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. *Proceedings of the 34th International Conference on Machine Learning*, pages 1809–1818.

[Kearns and Singh, 1998] Kearns, M. and Singh, S. (1998). Near-optimal reinforcement learning in polynomial time. *Proceedings of the 15th International Conference on Machine Learning*.

[Kearns and Koller, 1999] Kearns, M. J. and Koller, D. (1999). Efficient reinforcement learning in factored MDPs. *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 740–747.

[Kendall and Gal, 2017] Kendall, A. and Gal, Y. (2017). What uncertainties do we need in Bayesian deep learning for computer vision? *CoRR abs/1703.04977*.

[Kingma et al., 2015] Kingma, D., Salimans, T., and Welling, M. (2015). Variational dropout and the local reparameterization trick. *Proceedings of the 25th Advances in Neural Information Processing Systems*.

[K.Kiguchi et al., 2003] K.Kiguchi, T.Nanayakkara, K.Watanabe, and T.Fukuda (2003). Multi-dimensional reinforcement learning using a vector Q-net - application to mobile robots. *Internal Journal of Control, Automation and Systems*, 1:142–148.

[Konidaris and Barto, 2006] Konidaris, G. and Barto, A. (2006). Autonomous shaping: Knowledge transfer in reinforcement learning. *Proceedings of the 23rd International Conference on Machine Learning*, pages 489–496.

[Korf, 1985] Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109.

[Korf, 1997] Korf, R. E. (1997). Finding optimal solutions to rubik's cube using pattern databases. *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 700–705.

[Korf and Felner, 2002] Korf, R. E. and Felner, A. (2002). Disjoint pattern database heuristics. *Artificial Intelligence*, 22:9–22.

[Korf and Taylor, 1996] Korf, R. E. and Taylor, L. A. (1996). Finding optimal solutions to the twenty-four puzzle. *Proceedings of the 13th National Conference on Artificial Intelligence*, pages 1202–1207.

[Li, 2009] Li, L. (2009). *A Unifying Framework for Computational Reinforcement Learning Theory*. PhD thesis, Rutgers The State University of New Jersey-New Brunswick.

[Li et al., 2008] Li, L., Littman, M. L., and Walsh, T. J. (2008). Knows what it knows: A framework for self-aware learning. *Proceedings of the 25th International Conference on Machine Learning*, pages 568–575.

[Magriel and Roberts, 2004] Magriel, P. and Roberts, R. M. (2004). *Backgammon*. Clock & Rose Press.

[Marom and Rosman, 2018a] Marom, O. and Rosman, B. S. (2018a). Belief reward shaping in reinforcement learning. *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*.

[Marom and Rosman, 2018b] Marom, O. and Rosman, B. S. (2018b). Zero-shot transfer with deictic object-oriented representation in reinforcement learning. *Proceedings of the 32nd Conference on Neural Information Processing Systems*.

[Marom and Rosman, 2020] Marom, O. and Rosman, B. S. (2020). Utilising uncertainty for efficient learning of likely-admissible heuristics. *Proceedings of the 30th International Conference on Automated Planning and Scheduling*.

[Mataric, 1994] Mataric, M. (1994). Reward functions for accelerated learning. *Proceedings of the 11th International Conference on Machine Learning*, pages 181–189.

[Matignon et al., 2006] Matignon, L., Laurent, G. J., and le Fort-Piat, N. (2006). Reward function and initial values: Better choices for accelerated goal-directed reinforcement learning. *Lecture Notes in Computer Science*, 4131:840–849.

[McAleer et al., 2018] McAleer, S., Agostinelli, F., Shmakov, A., and Baldi, P. (2018). Solving the rubik's cube without human knowledge. *CoRR abs/1805.07470*.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with deep reinforcement learning. *CoRR abs/1312.5602*.

[Murphy, 2012] Murphy, K. P. (2012). *Machine Learning A Probabilistic Perspective*. MIT Press.

[Narvekar et al., 2020] Narvekar, S., Peng, B., Leonetti, M., Sinapov, J., Taylor, M. E., and Stone, P. (2020). Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research*, 21:1–50.

[Ng et al., 1999] Ng, Y., Harada, D., and Russell, S. J. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. *Proceedings of the 16th International Conference on Machine Learning*, pages 278–287.

[Osband, 2016] Osband, I. (2016). Risk versus uncertainty in deep learning: Bayes, bootstrap and the dangers of dropout. *Workshop on Bayesian Deep Learning at Advances in Neural Information Processing System*.

[Pan and Yang, 2010] Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22:1345–1359.

[Pearl, 1984] Pearl, J. (1984). *Heuristics*. Addison-Wesley.

[Perkins and Salomon, 1992] Perkins, D. N. and Salomon, G. (1992). Transfer of learning. *International Encyclopedia of Education*.

[Randløv and Alstrøm, 1998] Randløv, J. and Alstrøm, P. (1998). Learning to drive a bicycle using reinforcement learning and shaping. *Proceedings of the 15th International Conference on Machine Learning*, pages 463–471.

[Rosman and Ramamoorthy, 2012] Rosman, B. and Ramamoorthy, S. (2012). What good are actions? Accelerating learning using learned action priors. *Development and Learning and Epigenetic Robotics 2012 IEEE International Conference*.

[Rumelhart et al., 1986] Rumelhart, D. E., Hintont, G. E., and Williams., R. J. (1986). Learning representations by backpropagating errors. *Nature*, 323(6088):533–536.

[Russell and Norvig, 1995] Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

[Samadi et al., 2008] Samadi, M., Felner, A., and Schaeffer, J. (2008). Learning from multiple heuristics. *Proceedings of the 23rd National Conference on Artificial Intelligence*, pages 357–362.

[Scholz et al., 2014] Scholz, J., Levihn, M., Isbell, C. L., and Wingate, D. (2014). A physics-based model prior for object-oriented MDPs. *Proceedings of the 31st International Conference on Machine Learning*, pages 1089–1097.

[Shridhar et al., 2019] Shridhar, K., Laumann, F., and Liwicki, M. (2019). A comprehensive guide to Bayesian convolutional neural network with variational inference. *CoRR abs/1901.02731*.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489.

[Silver et al., 2017] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR/abs/1712.01815*.

[Singh et al., 2004] Singh, S., Barto, A., and Chentanez, N. (2004). Intrinsically motivated reinforcement learning. *Proceedings of the 18th Conference on Neural Information Processing Systems*.

[Slaney and Thiébaux, 2001] Slaney, J. and Thiébaux, S. (2001). Blocks world revisited. *Artificial Intelligence*, 125:119–153.

[Srivastava et al., 2014] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

[Strehl, 2007] Strehl, A. L. (2007). Model-based reinforcement learning in factored-state MDPs. *Proceedings of the 2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*.

[Strehl et al., 2007] Strehl, A. L., Diuk, C., and Littman, M. L. (2007). Efficient structure learning in factored state MDPs. *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pages 645–650.

[Strehl et al., 2006] Strehl, A. L., Li, L., Wiewiora, E., Langford, J., and Littman, M. L. (2006). PAC model-free reinforcement learning. *Proceedings of the 23rd International Conference on Machine Learning*, pages 881–888.

[Strehl and Littman, 2005] Strehl, A. L. and Littman, M. L. (2005). A theoretical analysis of model-based interval estimation. *Proceedings of the 22rd International Conference on Machine Learning*, pages 857–864.

[Strehl and Littman, 2008] Strehl, A. L. and Littman, M. L. (2008). Online linear regression and its application to model-based reinforcement learning. *Proceedings of the 21st Annual Conference on Neural Information Processing Systems*.

[Sun et al., 2017] Sun, S., Chen, C., and Carin, L. (2017). Learning structured weight uncertainty in Bayesian neural networks. *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, pages 1283–1292.

[Sutton et al., 1999] Sutton, R., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211.

[Sutton, 1990] Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the 7th international conference on machine learning*, pages 216–224.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.

[Taylor and Stone, 2009] Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685.

[Tesauro, 1995] Tesauro, G. J. (1995). Temporal difference learning and TD-Gammon. *Communication of the ACM*, 38:58–68.

[Tesauro, 2004] Tesauro, G. J. (2004). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219.

[Thayer et al., 2011] Thayer, J. T., Dionne, A. J., and Ruml, W. (2011). Learning inadmissible heuristics during search. *Proceedings of the 21st International Conference on Automated Planning and Scheduling*.

[van Otterlo, 2005] van Otterlo, M. (2005). A survey of reinforcement learning in relational domains. *CTIT Technical Report Series*.

[van Seijen et al., 2017] van Seijen, H., Fatemi, M., Romoff, J., Laroche, R., Barnes, T., and Tsang, J. (2017). Hybrid reward architecture for reinforcement learning. *CoRR abs/1706.04208*.

[Watkins, 1989] Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College.

[Wiewiora et al., 2003] Wiewiora, E., Cottrel, G. W., and Elkan, C. (2003). Principled methods for advising reinforcement learning agents. *Proceedings of the 20th International Conference on Machine Learning*, pages 792–799.

[Wingate, 2004] Wingate, D. (2004). Solving large MDPs quickly with partitioned value iteration. Master's thesis, Brigham Young University.

[Wingate and Seppi, 2004] Wingate, D. and Seppi, K. D. (2004). A partitioned, prioritized, parallel value iterator. *Proceedings of the 21st international conference on Machine learning*.

[Woof and Chen, 2018] Woof, W. and Chen, K. (2018). Learning to play general video-games via an object embedding network. *Proceedings of the 2018 IEEE Conference on Computational Intelligence and Games*, pages 1–8.

[Yang et al., 2008] Yang, F., Culberson, J., Holte, R., Zahavi, U., and Felner, A. (2008). A general theory of additive state space abstractions. *Artificial Intelligence*, 32:631–662.

[Yitao et al., 2016] Yitao, L., Machado, M. C., Talvitie, E., and Bowling, M. (2016). State of the art control of Atari games using shallow reinforcement learning. *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems*, pages 485–493.

[Zahavi et al., 2008] Zahavi, U., Felner, A., Holte, R., and Schaeffer, J. (2008). Duality in permutation state spaces and the dual search algorithm. *Artificial Intelligence*, 172(4-5):514–540.

Appendix A

Belief Reward Shaping

A.1 Cliff-Jump Gridworld

This section tabularises the average number of steps to complete an episode for each of the algorithms and parameter settings in the cliff-jump gridworld task discussed in Section 3.4.1. Results are averaged over 50 independent runs, with each run learning over 1000 episodes. Results are rounded to 1 decimal place and we highlight the best performing parameter setting for each algorithm in the tables.

Table A.1 shows results with standard Q-learning; Table A.2 shows results with Belief Reward Shaping (BRS); Tables A.3 and A.4 show results with Potential-Based Reward Shaping (PBRS) for positive and negative potentials respectively; and Tables A.5 and A.6 show results with Potential-Based Advice (PBA) for positive and negative potentials respectively.

Average	Standard Deviation
59.6	26.2

Table A.1: Cliff-jump performance with standard Q-learning.

A.2 Backgammon

This section describes the belief clusters and parameters used in the backgammon experiment described in Section 3.4.3. Table A.8 describes the setup with simple prior beliefs; Table A.9 describes the setup with complex prior beliefs; and Table A.7 describes the abbreviations for the game state positions used for the belief clusters in Tables A.8 andA.9.

μ_0, λ	Average	Standard Deviation
$0.5, 10^2$	55.8	24.7
$1, 10^2$	52.0	23.9
$5, 10^2$	39.0	14.5
$10, 10^2$	30.9	3.4
$100, 10^2$	34.5	4.9
$0.5, 10^3$	41.4	15.9
$1, 10^3$	31.0	4.1
$5, 10^3$	31.3	2.9
$10, 10^3$	32.4	2.7
$100, 10^3$	66.1	29.4
$0.5, 10^4$	30.4	3.0
$1, 10^4$	**30.2**	**2.5**
$5, 10^4$	32.9	5.8
$10, 10^4$	38.9	11.4
$100, 10^4$	58.8	30.5

Table A.2: Cliff-jump performance with BRS.

μ_0	Average	Standard Deviation
0.5	**48.4**	**21.6**
1	49.8	17.5
5	62.0	10.5
10	61.7	10.5
100	62.8	9.9

Table A.3: Cliff-jump performance with PBRS and positive potentials.

For the simple set of prior beliefs in Table A.8 the criteria are chosen so that the positions are 'improvements' over the starting position as at the start of a game each side has 0 blots, 1 inner point, 4 points and a maximum prime of 1. The complex set of prior beliefs are more advanced because they scale the reward based on the position (i.e. having 5 blot is considered worse than 2 blots) and restrict certain rewards to situations when the position is beneficial (i.e. there is an opponent checker in the player's inner board or on the bar). For the complex set of priors each i in Table A.9 is a separate belief cluster with its own prior mean and prior mean observations.

Furthermore, note that that the neural network has 4 output neurons to account for gammon / backgammon wins and losses. Therefore, in these tables, a value of $\mu_0 = c$ corresponds to a vector $[c, c, -c, -c]$, where the first two elements represents a win for player 1 and the last two elements represent wins for player 2. This vector is then used as the belief reward when training the network.

μ_0	Average	Standard Deviation
0.5	**54.1**	**24.2**
1	66.6	20.8
5	82.7	15.1
10	91.3	11.3
100	99	1.1

Table A.4: Cliff-jump performance with PBRS and negative potentials.

μ_0, μ_1	Average	Standard Deviation
$0.5, 0.5$	48.0	21.9
$1, 0.5$	46.9	19.8
$5, 0.5$	62.5	10.6
$10, 0.5$	61.4	10.4
$100, 0.5$	63.9	10.1
$0.5, 1$	48.1	21.7
$1, 1$	46.7	20.2
$5, 1$	62.9	10.6
$10, 1$	62.0	10.3
$100, 1$	63.9	10.2
$0.5, 5$	45.5	22.6
$1, 5$	44.9	21.9
$5, 5$	62.9	11.3
$10, 0.5$	61.9	10.6
$100, 0.5$	63.9	10.2
0.5,10	**43.2**	**24.7**
$1, 10$	43.4	25.7
$5, 10$	64.8	12.4
$10, 10$	62.6	10.8
$100, 10$	64.1	10.4
$0.5, 100$	92.1	10.2
$1, 100$	84.4	18.8
$5, 100$	78.9	18.3
$10, 100$	71.7	17.2
$100, 100$	65.5	12.1

Table A.5: Cliff-jump performance with PBA with positive potentials.

μ_0, μ_1	Average	Standard Deviation
0.5, 0.5	54.3	24.1
1, 0.5	62.6	20.5
5, 0.5	83.2	14.9
10, 0.5	91.7	10.4
100, 0.5	99.2	0.9
0.5, 1	53.4	24.9
1, 1	63.8	19.6
5, 1	83.4	14.8
10, 1	91.2	11.2
100, 1	98.9	1.1
0.5, 5	51.4	24.7
1, 5	61.1	21.5
5, 5	82.7	15.1
10, 0.5	92.1	10.3
100, 0.5	99.1	0.9
0.5,10	**49.7**	**24.9**
1, 10	58.6	22.7
5, 10	82.3	15.5
10, 10	90.7	11.5
100, 10	99.0	1.0
0.5, 100	63.8	19.9
1, 100	67.9	17.2
5, 100	82.6	14.1
10, 100	89.7	11.9
100, 100	99.3	0.9

Table A.6: Cliff-jump performance with PBA and negative potentials.

Abbreviation	Description
dc	degree of contact
p1_bl	player 1 number of blots
p2_bl	player 2 number of blots
p1_ipt	player 1 number of inner points
p2_ipt	player 2 number of inner points
p1_pt	player 1 number of points
p2_pt	player 2 number of points
p1_pr	player 1 maximum prime
p2_pr	player 2 maximum prime
p1_iopp	player 2 checkers in player 1 inner board / on bar
p2_iopp	player 1 checkers in player 2 inner board / on bar

Table A.7: Backgammon game state position abbreviations for the belief clusters.

Belief Cluster	μ_0	λ
dc>0, p1_bl>0	-0.5	3000
dc>0 ,p2_bl>0	0.5	3000
dc>0, p1_ipt>1	0.5	3000
dc>0, p2_ipt>1	-0.5	3000
dc>0, p1_pt>4	0.5	3000
dc>0, p2_pt>4	-0.5	3000
dc>0, p1_pr>1	0.5	3000
dc>0, p2_pr>1	-0.5	3000

Table A.8: Backgammon simple set of prior beliefs.

Belief Cluster	μ_0	λ	Range
dc>0, p1_bl=i	-1.5 $\frac{Min(i,5)}{15}$	3000	$i \in [0, 15]$
dc>0, p2_bl=i	1.5 $\frac{Min(i,5)}{15}$	3000	$i \in [0, 15]$
dc>0, p1_iopp>0, p1_ipt=i	0.5 $\frac{i}{6}$	3000	$i \in [0, 6]$
dc>0, p2_iopp>0, p2_ipt=i	-0.5 $\frac{i}{6}$	3000	$i \in [0, 6]$
dc>0, p1_pt=i	0.5 $\frac{i}{7}$	3000	$i \in [0, 7]$
dc>0, p2_pt=i	-0.5 $\frac{i}{7}$	3000	$i \in [0, 7]$
dc>0, p1_iopp>0, p1_pr=i	0.5 $\frac{i}{7}$	3000	$i \in [0, 7]$
dc>0, p2_iopp>0, p2_pr=i	-0.5 $\frac{i}{7}$	3000	$i \in [0, 7]$

Table A.9: Backgammon complex set of prior beliefs.

Appendix B

Zero-Shot Transfer with Object-Oriented Representations

B.1 All-Passenger Any-Destination Taxi Domain

In Chapter 4 we introduce the All-Passenger Any-Destination Taxi domain. In this section, we outline the representation of the transition dynamics and reward dynamics under the object-oriented formalisms presented in that chapter. We also provide further details on the experimental setup for this domain.

B.1.1 Transition Dynamics

For the transition dynamics define the following deictic predicates.

- *AnyWallNorthOfTaxi*($taxi, s$): returns 1 if there is any object of class *Wall* one square north of *taxi* in s, and 0 otherwise. (f_1)
- *AnyWallEastOfTaxi*($taxi, s$): returns 1 if there is any obect of class *Wall* one square east of *taxi* in s, and 0 otherwise. (f_2)
- *AnyWallSouthOfTaxi*($taxi, s$): returns 1 if there is any object of class *Wall* one square south of *taxi* in s, and 0 otherwise. (f_3)
- *AnyWallWestOfTaxi*($taxi, s$): returns 1 if there is object of class *Wall* one square west of *taxi* in s, and 0 otherwise. (f_4)
- *AnyTaxiOnPassenger*($passenger, s$): returns 1 if there is any object of class *Taxi* on the same square as *passenger* in s, and 0 otherwise. (f_5)
- *PassengerAtAnyDestination(passenger,s)*: returns 1 if *passenger.at-destination* is set to 1 in s, and 0 otherwise. (f_6)

- *AnyPassengerInAnyTaxi*(*passenger*, s): returns 1 if any object of class *Passenger* has its *in-taxi* attribute is set to 1 in s, and 0 otherwise. (f_7)
- *PassengerInAnyTaxi*(*passenger*, s): returns 1 if *passenger.in-taxi* is set to 1 in s, and 0 otherwise. (f_8)
- *AnyTaxiOnAnyDestination*(*destination*, s): returns 1 if any object of class *Taxi* is on the same square as any object of class *Destination* in s, and 0 otherwise. (f_9)

As explained in Section 4.4.2, *AnyPassengerInAnyTaxi*(*passenger*, s) and *AnyTaxiOnAnyDestination*(*destination*, s) are actually propositions and the deictic objects are included for notational consistency of the formalism.

Define the following effect sets:

- $Rel_i(x) = x + i$ for $i \in \{-1, 0, 1\}$ where x is an integer.
- $SetBool_i(x) = x\mathbb{1}(i = 0) + (1 - x)\mathbb{1}(i = 1)$ for $i \in \{0, 1\}$ where $x \in \{0, 1\}$.

Then Table B.1 shows for each attribute and action the relevant preconditions and effects that describe the transition dynamics of the domain.

Action	Attribute	Precondition	Effect
North	*Taxi.y*	$f_1 = 0$	Rel_1
North	*Taxi.y*	$f_1 = 1$	Rel_0
East	*Taxi.x*	$f_2 = 0$	Rel_1
East	*Taxi.x*	$f_2 = 1$	Rel_0
South	*Taxi.y*	$f_3 = 0$	Rel_{-1}
South	*Taxi.y*	$f_3 = 1$	Rel_0
West	*Taxi.x*	$f_4 = 0$	Rel_{-1}
West	*Taxi.x*	$f_4 = 1$	Rel_0
Pickup	*Passenger.in-taxi*	$f_5 = 1 \wedge f_6 = 0 \wedge f_7 = 0$	$SetBool_1$
Pickup	*Passenger.in-taxi*	$f_5 = 0 \vee f_6 = 1 \vee f_7 = 1$	$SetBool_0$
Dropoff	*Passenger.in-taxi*	$f_8 = 1 \wedge f_9 = 1$	$SetBool_1$
Dropoff	*Passenger.in-taxi*	$f_8 = 0 \vee f_9 = 0$	$SetBool_0$
Dropoff	*Passenger.at-destination*	$f_8 = 1 \wedge f_9 = 1$	$SetBool_1$
Dropoff	*Passenger.at-destination*	$f_8 = 0 \vee f_9 = 0$	$SetBool_0$

Table B.1: Precondition and effect for each action and attribute in the All-Passenger Any-Destination Taxi domain. We exclude *Wall* attributes because they never change.

B.1.2 Reward Dynamics

The reward dynamics for the All-Passenger Any-Destination Taxi domain require $L = 3$ reward tokens in addition to the default reward token. Define the following

propositions:

- *AnyTaxiOnAnyPassenger*(s, s'): returns 1 if any objects of class *Taxi* is on the same square as any object of class *Passenger* in s, and otherwise 0. (p_1)
- *AnyPassengerInAnyTaxi*(s, s'): returns 1 if any objects of class *Passenger* has their *in-taxi* attribute set to 1 in s, and otherwise 0. (p_2)
- *AnyTaxiOnAnyDestination*(s, s'): returns 1 if any objects of class *Taxi* is on the same square as any object of class *Destination* in s, and otherwise 0. (p_3)
- *AllPassenagersAtAnyDestination*(s, s'): returns 1 if all objects of class *Passenger* have their *at-destination* attributes set to 1 in s', and otherwise 0. (p_4)

Define the following reward tokens:

- $U_1 = -10$
- $U_2 = -10$
- $U_3 = 0$
- $U_4 = -1$

Then Table B.2 shows the reward token that activates for each action and precondition.

Action	Precondition	Reward Token
Pickup	$p_1 = 0 \vee p_1 = 1 \wedge p_2 = 1$	U_1
Dropoff	$p_2 = 0 \vee p_2 = 1 \wedge p_3 = 0$	U_2
Dropoff	$p_4 = 1$	U_3

Table B.2: Reward tokens that activate for each action and precondition in the All-Passenger Any-Destination Taxi domain. Any other combination leads to the default reward token U_4.

B.1.3 Experiment for Learning Transition Dynamics

In Section 4.7.1 we run experiments on the All-Passenger Any-Destination Taxi domain to learn transition dynamics. In this section we describe the hypothesised deictic predicates used for each action and attribute in the experiments. In these experiments we compare results to Propositional OO-MDPs. In doing so, we need to choose a set of hypothesized propositions for the Propositional OO-MDP part of the experiments as well. In constructing our experiments, we set out to mimic as closely as possible the representation described for the classical Taxi domain under

Propositional OO-MDPs. In particular, for the classic Taxi domain, the work on Propositional OO-MDPs hypothesised the following proposition for every action and attribute [Diuk et al., 2008]:

- $Touch_N(Taxi, Wall)$: returns 1 if any object of class *Taxi* has an object of class *Wall* one square north of it, and 0 otherwise. (P_1)
- $Touch_E(Taxi, Wall)$: returns 1 if any object of class *Taxi* has an object of class *Wall* one square east of i, and 0 otherwise. (P_2)
- $Touch_S(Taxi, Wall)$: returns 1 if any object of class *Taxi* has an object of class *Wall* one square south of it, and 0 otherwise. (P_3)
- $Touch_W(Taxi, Wall)$: returns 1 if any object of class *Taxi* has an object of class *Wall* one square west of it, and 0 otherwise. (P_4)
- $On(Taxi, Passenger)$: returns 1 if any object of class *Taxi* is on the same square as any object of class *Passenger*, and 0 otherwise. (P_5)
- $On(Taxi, Destination)$: returns 1 if any object of class *Taxi* is on the same square as any object of class *Destination*, and 0 otherwise. ($P_6 = f_9$)
- $InTaxi(Passenger)$: returns 1 if any object of class *Passenger* has its *in-taxi* attribute set to 1, and 0 otherwise. ($P_7 = f_7$)

In order to remain comparable with Propositional OO-MDPs, our approach is to use deictic predicates when we can while falling back on using propositions when we must. For example, consider the *Passenger.in-taxi* attribute. It is not possible to convert $Touch_N(Taxi, Wall)$ to a deictic predicate since when predicting the *Passenger.in-taxi* attribute we only have access to a grounded *passenger* object, and we do not have access to any grounded *wall* or *taxi* objects. So, in this case, we use propositions. However we can switch $On(Taxi, Passenger)$ to a deictic predicate $AnyTaxiOnAnyPassenger(passenger, s)$ relative to a deictic *passenger* object. We note that for the one-passenger experiments we have similar performance between Propositional OO-MDPs and Deictic OO-MPDs as shown in Figure 4.8, indicating that we have not biased Deictic-OO MDPs in our experiments. Our experiments then use the setup as shown in Table B.3.

When running $DOORMAX_D$, we take advantage of conjunctive effects to invalidate multiple terms with single observations. This is the core mechanism for efficient learning under the algorithm. To illustrate this process, we show in tables B.4 and B.5 how the algorithm converges to the true precondition from a simulation of the algorithm. Table B.4 is a simulation for action *East*, attribute *Taxi.x* and effect Rel_1; Table B.5 is a simulation for action *Pickup*, attribute *Passenger.in-taxi* and

Action	Attribute	Hypothesis	Conjunctive
Any	*Taxi.x*	$\{f_1, f_2, f_3, f_4, P_5, P_6, P_7\}$	$\{-1, 0, 1\}$
Any	*Taxi.y*	$\{f_1, f_2, f_3, f_4, P_5, P_6, P_7\}$	$\{-1, 0, 1\}$
Pickup	*Passenger.in-taxi*	$\{f_5, f_6, P_1, P_2, P_3, P_4, P_7\}$	$\{1\}$
Dropoff	*Passenger.in-taxi*	$\{f_8, P_1, P_2, P_3, P_4, P_5, P_6\}$	$\{1\}$
Dropoff	*Passenger.at-destination*	$\{f_8, P_1, P_2, P_3, P_4, P_5, P_6\}$	$\{1\}$

Table B.3: Hypothesised deictic predicates / propositions for each action and attribute as well as conjunctive effects. Any other action and attribute uses the hypothesis $\{P_1, P_2, P_3, P_4, P_5, P_6, P_7\}$ with all effects being conjunctive.

effect $SetBool_1$. Note that in Table B.4 we are able to learn the correct precondition with only 7 observations, whereas learning by memorisation would require 2^6 unique observations.

observation	f_1	f_2	f_3	f_4	P_5	P_6	P_7
1	0	0	0	1	0	0	0
2	-	0	0	1	0	-	0
3	-	0	0	-	0	-	0
4	-	0	0	-	0	-	-
5	-	0	0	-	0	-	-
6	-	0	0	-	0	-	-
7	-	0	-	-	-	-	-

Table B.4: Learning the precondition for action *East*, attribute *Taxi.x* and effect Rel_1 in the All-Passenger Any-Destination Taxi domain.

observation	P_1	P_2	P_3	P_4	f_5	f_6	P_7
1	0	1	1	0	1	0	0
2	-	1	-	0	1	0	0
3	-	-	-	-	1	0	0

Table B.5: Learning the precondition for action *Pickup*, attribute *Passenger.in-taxi* and effect $SetBool_1$ in the All-Passenger Any-Destination Taxi domain.

B.1.4 Experiment for Learning Reward Dynamics

In Section 4.7.2 we run experiments on the All-Passenger Any-Destination Taxi domain to learn reward dynamics. In this section we describe the hypothesised propositions used in the experiments.

In additon to p_1, p_2, p_3 and p_4 defined in Section B.1.2 we define:

- *AnyWallNorthOfAnyTaxi*(s, s'): returns 1 if any objects of class *Taxi* has an object of class *Wall* one square north of it in s', and otherwise 0. (p_5)
- *AnyWallEastOfAnyTaxi*(s, s'): returns 1 if any objects of class *Taxi* has an object of class *Wall* one square east of it in s', and otherwise 0. (p_6)
- *AnyWallSouthOfAnyTaxi*(s, s'): returns 1 if any objects of class *Taxi* has an object of class *Wall* one square south of it in s', and otherwise 0. (p_7)
- *AnyWallWestOfAnyTaxi*(s, s'): returns 1 if any objects of class *Taxi* has an object of class *Wall* one square west of it in s', and otherwise 0. (p_8)

Then in our experiments we use the following hypothesis in Table B.6 for each action and reward token.

Action	Reward Token	Hypothesis	Conjunctive
Pickup	U_1	$\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$	$\{0\}$
Dropoff	U_2	$\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$	$\{0\}$
Dropoff	U_3	$\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$	$\{0, 1\}$

Table B.6: Hypothesised propositions for each action and reward token as well as their conjunctive indicator values. All other actions and reward token pairs use the hypothesis $\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$ with conjunctive indicators $\{0, 1\}$. The default reward token is U_4.

B.2 Sokoban

In Section 4.7.3 we run experiments on the Sokoban domain showing that zero-shot transfer of both transition and reward dynamics is possible under our proposed framework. In this section we provide more details on the setup of these under an object-oriented approach as described in Chapter 4.

B.2.1 Transition Dynamics

In this section we show for each attribute and action the relevant precondition and effect for the Sokoban domain under the Deictic OO-MDP formalism. Note that in Sokoban it is possible to get stuck in a state from which the task is no longer solvable – this is called a *deadlock* state. To manage this, we add a *Reset* action that sets a *person* object's *reset* attribute to 1 and that immediately terminates the task. As further discussed in Section B.2.2, for this domain, we use a reward of 300 for getting all the boxes to a storage location, -1 for activating the *reset* attribute and -1 for any navigation step. The high reward for getting all boxes to a storage

location is necessary to prevent the agent from learning an optimal policy whereby it simply applies the *Reset* action at the start of every task.

We define the following deictic predicates for the transitions dynamics:

- $AnyWallNorthOfPerson_i(person, s)$: returns 1 if there is any object of class *Wall* i squares north of *person* in state s, and 0 otherwise. ($f_1(i)$)
- $AnyWallEastOfPerson_i(person, s)$: returns 1 if there is any object of class *Wall* i squares east of *person* in state s, and 0 otherwise. ($f_2(i)$)
- $AnyWallSouthOfPerson_i(person, s)$: returns 1 if there is any object of class *Wall* i squares south of *person* in state s, and 0 otherwise. ($f_3(i)$)
- $AnyWallWestOfPerson_i(person, s)$: returns 1 if there is any object of class *Wall* i squares west of *person* in state s, and 0 otherwise. ($f_4(i)$)
- $AnyBoxNorthOfPerson_i(person, s)$: returns 1 if there is any object of class *Box* i squares north of *person* in state s, and 0 otherwise.($f_5(i)$)
- $AnyBoxEastOfPerson_i(person, s)$: returns 1 if there is any object of class *Box* i squares east of *person* in state s, and 0 otherwise. ($f_6(i)$)
- $AnyBoxSouthOfPerson_i(person, s)$: returns 1 if there is any object of class *Box* i squares south of *person* in state s, and 0 otherwise. ($f_7(i)$)
- $AnyBoxWestOfPerson_i(person, s)$: returns 1 if there is any object of class *Box* i squares west of *person* in state s, and 0 otherwise. ($f_8(i)$)
- $AnyPersonNorthOfBox(box, s)$: returns 1 if there is any object of class *Person* one square north of *box* in s, and otherwise 0. (f_{10})
- $AnyPersonEastOfBox(box, s)$: returns 1 if there is any object of class *Person* one square east of *box* in s, and otherwise 0. (f_{11})
- $AnyPersonSouthOfBox(box, s)$: returns 1 if there is any object of class *Person* one square south of *box* in s, and otherwise 0. (f_{12})
- $AnyPersonWestOfBox(box, s)$: returns 1 if there is any object of class *Person* one square west of *box* in s, and otherwise 0. (f_{13})
- $AnyBoxNorthOfBox(box, s)$: returns 1 if there is any object of class *Box* one square north of *box* in s, and otherwise 0. (f_{14})
- $AnyBoxEastOfBox(box, s)$: returns 1 if there is any object of class *Box* one square east of *box* in s, and otherwise 0. (f_{15})

- *AnyBoxSouthOfBox*(box, s): returns 1 if there is any object of class *Box* one square south of *box* in s, and otherwise 0. (f_{16})
- *AnyBoxWestOfBox*(box, s): returns 1 if there is any object of class *Box* one square west of *box* in s, and otherwise 0. (f_{17})
- *AnyWallNorthOfBox*(box, s): returns 1 if there is any object of class *Wall* one square north of *box* in s, and otherwise 0. (f_{18})
- *AnyWallEastOfBox*(box, s): returns 1 if there is any object of class *Wall* one square east of *box* in s, and otherwise 0. (f_{19})
- *AnyWallSouthOfBox*(box, s): returns 1 if there is any object of class *Wall* one square south of *box* in s, and otherwise 0. (f_{20})
- *AnyWallWestOfBox*(box, s): returns 1 if there is any object of class *Wall* one square west of *box* in s, and otherwise 0. (f_{21})
- *ResetActivated*($person, s$): returns 1 if $person.reset = 1$ in s, otherwise 0. (f_{22})

We define the following effect sets:

- $Rel_i(x) = x + i$ for $i \in \{-1, 0, 1\}$ where x is an integer
- $SetBool_i(x) = x\mathbb{1}(i = 0) + (1 - x)\mathbb{1}(i = 1)$ for $i \in \{0, 1\}$ where $x \in \{0, 1\}$.

Then Table B.7 shows for each action and attribute the relevant precondition and effects.

B.2.2 Reward Dynamics

For this domain we set $L = 2$. We define the following propositions for the reward dynamics:

- *AllBoxesAtAnyStorage*(s, s'): returns 1 if all objects of class *Box* are on the same square as any object of class *Storage* in s', and otherwise 0. (p_1)
- *ResetActivated*(s, s'): returns 1 if any object of class *Person* has their *reset* attribute set to 1 in s', otherwise 0. (p_2)

Define the following reward tokens:

- $U_1 = 300$
- $U_2 = -1$
- $U_3 = -1$

Action	Attribute	Precondition	Effect
North	*Person.y*	$f_1(1) = 0 \wedge f_5(1) = 0 \vee f_5(1) = 1 \wedge f_1(2) = 0 \wedge f_5(2) = 0$	Rel_1
North	*Person.y*	$f_1(1) = 1 \vee f_5(1) = 1 \wedge f_1(2) = 1 \vee f_5(1) = 1 \wedge f_5(2) = 1$	Rel_0
East	*Person.x*	$f_2(1) = 0 \wedge f_6(1) = 0 \vee f_6(1) = 1 \wedge f_2(2) = 0 \wedge f_6(2) = 0$	Rel_1
East	*Person.x*	$f_2(1) = 1 \vee f_6(1) = 1 \wedge f_2(2) = 1 \vee f_6(1) = 1 \wedge f_6(2) = 1$	Rel_0
South	*Person.y*	$f_3(1) = 0 \wedge f_7(1) = 0 \vee f_7(1) = 1 \wedge f_3(2) = 0 \wedge f_7(2) = 0$	Rel_{-1}
South	*Person.y*	$f_3(1) = 1 \vee f_7(1) = 1 \wedge f_3(2) = 1 \vee f_7(1) = 1 \wedge f_7(2) = 1$	Rel_0
West	*Person.x*	$f_4(1) = 0 \wedge f_8(1) = 0 \vee f_8(1) = 1 \wedge f_4(2) = 0 \wedge f_8(2) = 0$	Rel_{-1}
West	*Person.x*	$f_4(1) = 1 \vee f_8(1) = 1 \wedge f_4(2) = 1 \vee f_8(1) = 1 \wedge f_8(2) = 1$	Rel_0
North	*Box.y*	$f_{12} = 1 \wedge f_{14} = 0 \wedge f_{18} = 0$	Rel_1
North	*Box.y*	$f_{12} = 0 \vee f_{14} = 1 \vee f_{18} = 1$	Rel_0
East	*Box.x*	$f_{13} = 1 \wedge f_{15} = 0 \wedge f_{19} = 0$	Rel_1
East	*Box.x*	$f_{13} = 0 \vee f_{15} = 1 \vee f_{19} = 1$	Rel_0
South	*Box.y*	$f_{10} = 1 \wedge f_{16} = 0 \wedge f_{20} = 0$	Rel_{-1}
South	*Box.y*	$f_{10} = 0 \vee f_{16} = 1 \vee f_{20} = 1$	Rel_0
West	*Box.x*	$f_{11} = 1 \wedge f_{17} = 0 \wedge f_{21} = 0$	Rel_{-1}
West	*Box.x*	$f_{11} = 0 \vee f_{17} = 1 \vee f_{21} = 1$	Rel_0
Reset	*Person.reset*	$f_{22} = 1$	$SetBool_1$

Table B.7: Precondition and effect for each action and attribute in the Sokoban domain. We exclude *Wall* attributes since they cannot change.

Then Table B.8 shows the reward token that activates for each action and precondition.

Action	Precondition	Reward Token
North	$p_1 = 1$	U_1
East	$p_1 = 1$	U_1
South	$p_1 = 1$	U_1
West	$p_1 = 1$	U_1
Reset	$p_2 = 1$	U_2

Table B.8: Reward tokens that activate for each action and precondition in the Sokoban domain. Any other combination leads to the default reward token U_3.

Appendix C

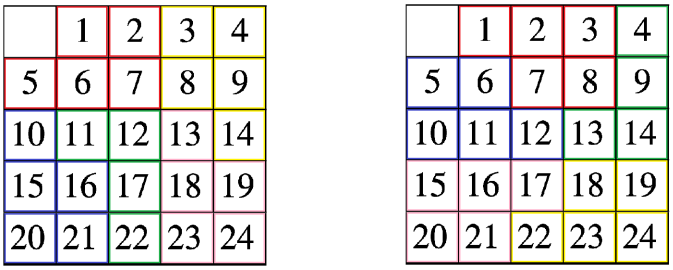
Efficient Learning of Likely-Admissible Heuristics

C.1 Pattern Databases

In Section 5.5 when running experiments on the 24-puzzle, 24-pancake and 15-blocksworld domains we make use of pattern databases (PDBs) as features for the neural network. In this section we detail the patterns used for each domain. The PDBs are described from the reference point of the goal state of each domain.

C.1.1 24-puzzle

For the 24-puzzle domain we use two sets of disjoint 5-5-5-4 PDBs.



(a) First set of disjoint PDBs. (b) Second set of disjoint PDBs.

Figure C.1: Disjoint PDBs for 24-puzzle.

First set's partition: $\{1, 2, 5, 6, 7\}$, $\{3, 4, 8, 9, 14\}$, $\{10, 15, 16, 20, 21\}$, $\{11, 12, 17, 22\}$, $\{13, 18, 23, 24\}$

Second set's partition: $\{1, 2, 3, 7, 8\}$, $\{5, 6, 10, 11, 12\}$, $\{15, 16, 17, 20, 21\}$, $\{4, 9, 13, 14\}$, $\{18, 19, 22, 23, 24\}$

C.1.2 24-pancake

For the 24-pancake domain we use two sets of location-based disjoint 5-5-5-4 PDBs.

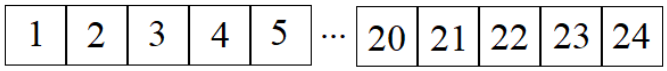


Figure C.2: Goal state for 24-pancake.

First set's partition: $\{1,2,3,4,5\}$, $\{6,7,8,9,10\}$, $\{11,12,13,14,15\}$, $\{16,17,18,19,20\}$, $\{21,22,23,24\}$.

Second set's partition: $\{1,2,3,19\}$, $\{4,5,6,7,8\}$, $\{9,10,11,12,13\}$, $\{14,15,16,17,18\}$, $\{20,21,22,23,24\}$.

C.1.3 15-blocksworld

For the 15-blocksworld domain we use 12 4-block PDBs.

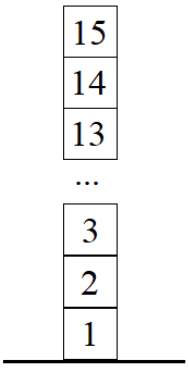


Figure C.3: Goal state for 15-blocksworld.

The 12 4-block PDBs are: $\{1,2,3,4\}$, $\{5,6,7,8\}$, $\{9,10,11,12\}$, $\{12,13,14,15\}$, $\{3,4,5,6\}$, $\{7,8,9,10\}$, $\{11,12,13,14\}$, $\{1,2,14,15\}$, $\{2,3,4,5\}$, $\{6,7,8,9\}$, $\{10,11,12,13\}$, $\{1,13,14,15\}$.

C.2 Detailed Results

In this section we include detailed tables for experiments run in Section 5.5 which include the standard deviation, in brackets. We note that the standard deviations for the statistics of the suboptimality experiments is very high for some domains (in some cases, higher than the means). This implies that the empirical distribution is highly skewed to the right. Finding techniques to reduce the run variance is an area of future research for learning likely-admissible heuristics under our framework.

Table C.1 shows detailed results for the 15-puzzle suboptimality experiments; Table C.2 shows detailed results for the 15-puzzle efficiency experiments; Table C.3 shows detailed results for the 24-puzzle suboptimality experiments; Table C.4 shows detailed results for the 24-pancake suboptimality experiments; Table C.5 shows detailed results for the 15-blocksworld suboptimality experiments; and Table C.6 shows detailed results for the runtime of the all the suboptimality experiments.

α	Time		Generated	Subopt		Optimal	
0.95	74.6	(81.50)	78,787,262 (76,296,444)	2.21%	(2.17%)	67.80%	(15.08%)
0.9	26.72	(25.44)	29,342,747 (27,830,962)	2.46%	(2.41%)	65.20%	(16.95%)
0.75	8.71	(11.40)	9,357,055 (12,682,566)	2.97%	(2.72%)	59.00%	(18.35%)
0.5	5.06	(5.56)	5,284,645 (6,160,880)	3.35%	(2.62%)	52.30%	(15.96%)
0.25	4.85	(6.99)	5,107,840 (7,728,056)	4.45%	(3.03%)	38.30%	(16.96%)
0.1	3.89	(5.02)	4,285,483 (5,996,957)	5.32%	(3.39%)	30.70%	(16.11%)
0.05	3.80	(4.67)	4,189,753 (5,696,733)	5.63%	(3.03%)	25.3%	(11.27%)
N/A	2.25	(3.09)	3,071,956 (4,749,797)	10.75%	(3.23%)	10.90%	(5.54%)

Table C.1: Detailed suboptimality results for 15-puzzle.

LengthInc	Solved Train		Solved Test	
1	100%	(0.00%)	38.59%	(3.56%)
2	95.10%	(2.27%)	48.20%	(3.32%)
4	61.80%	(10.57%)	51.40%	(16.21%)
6	36.20%	(6.01%)	39.61%	(3.97%)
8	19.20%	(3.49%)	35.16%	(3.56%)
10	11.80%	(6.78%)	31.83%	(12.04%)
GTP	93.30%	(2.65%)	60.59%	(12.32%)

Table C.2: Detailed efficiency results for 15-puzzle.

α	Time	Generated	Subopt	Optimal
0.95	2,664.53 (3,062.59)	1,233,965,823 (1,322,894,069)	2.15% (0.65%)	28.80% (11.36%)
0.9	1,371.29 (1,292.45)	628,101,474 (539,128,139)	2.64% (0.66%)	20.00% (9.55%)
0.75	549.22 (549.44)	274,003,465 (259,744,778)	3.67% (0.68%)	5.20% (4.66%)
0.5	189.37 (131.53)	99,244,234 (70,575,145)	4.64% (0.54%)	1.20% (1.60%)
0.25	121.18 (66.23)	66,147,586 (35,047,947)	5.18% (0.58%)	0.00% (0.00%)
0.1	86.62 (50.78)	46,988,530 (28,554,357)	5.81% (0.51%)	0.00% (0.00%)
0.05	83.56 (37.19)	40,046,361 (18,032,980)	6.26% (0.55%)	0.00% (0.00%)
N/A	25.39 (20.83)	11,719,659 (9,552,893)	11.34% (0.85%)	0.00% (0.00%)

Table C.3: Detailed suboptimality results for 24-puzzle.

α	Time	Generated	Subopt	Optimal
0.95	364.58 (85.68)	104,132,601 (27,914,343)	1.09% (0.09%)	76.00% (1.26%)
0.9	198.56 (37.55)	54,089,822 (10,731,888)	1.27% (0.07%)	72.40% (1.96%)
0.75	54.24 (12.35)	13,001,211 (1,713,639)	1.85% (0.13%)	59.20% (2.99%)
0.5	20.42 (3.88)	4,530,281 (820,070)	2.17% (0.11%)	53.20% (3.25%)
0.25	11.66 (2.03)	2,511,066 (396,223)	3.53% (0.29%)	37.20% (6.52%)
0.1	8.30 (3.75)	1,621,775 (843,061)	3.83% (0.72%)	30.80% (7.33%)
0.05	4.96 (2.65)	871,908 (441,617)	4.03% (0.79%)	30.80% (7.65%)
N/A	0.85 (1.30)	210,622 (338,283)	10.58% (4.73%)	8.40% (12.86%)

Table C.4: Detailed suboptimality results for 24-pancake.

α	Time		Generated	Subopt		Optimal	
0.95	55.51	(11.49)	115,691,631 (7,070,181)	0.02%	(0.03%)	99.60%	(0.80%)
0.9	53.79	(11.78)	112,390,208 (9,764,255)	0.07%	(0.06%)	98.40%	(1.50%)
0.75	50.52	(11.89)	101,109,757 (15,266,842)	0.23%	(0.20%)	95.60%	(3.67%)
0.5	38.01	(15.44)	69,663,441 (19,064,929)	0.98%	(0.49%)	84.80%	(7.22%)
0.25	43.97	(34.56)	63,963,572 (44,432,088)	4.28%	(2.01%)	50.80%	(13.00%)
0.1	35.83	(21.90)	50,951,658 (25,855,679)	9.70%	(5.87%)	34.40%	(11.20%)
0.05	28.50	(19.06)	42,499,655 (28,081,066)	13.36%	(9.05%)	24.00%	(12.46%)
N/A	20.88	(22.20)	31,178,090 (32,600,259)	7.07%	(3.50%)	38.40%	(13.71%)

Table C.5: Detailed suboptimality results for 15-blocksworld.

Domain	plan with $\hat{y}$		plan with y^α	
15-puzzle	1.32	(0.16)	2.67	(0.17)
24-puzzle	6.03	(0.36)	20.52	(1.32)
24-pancake	2.34	(0.19)	15.85	(1.19)
15-blocksworld	4.38	(0.47)	6.54	(0.27)

Table C.6: Detailed training runtime in hours.