

https://dspace.org/

Accelerating decision making under partial observability using learned action priors

Mabena, Ntokozo

2017

https://hdl.handle.net/10539/24175

Mabena, Ntokozo (2017) Accelerating decision making under partial observability using learned action priors, University of the Witwatersrand, Johannesburg, <http://hdl.handle.net/10539/24175>

Downloaded from a DSpace repository. https://dspace.org/

Accelerating Decision Making Under Partial Observability Using Learned Action Priors

Ntokozo Mabena
494350

Supervisor: Benjamin Rosman
Co-supervisor: Pravesh Ranchod

Master of Science
School of Computer Science and Applied Mathematics
University of Witwatersrand
2017



Abstract

Partially Observable Markov Decision Processes (POMDPs) provide a principled mathematical framework allowing a robot to reason about the consequences of actions and observations with respect to the agent's limited perception of its environment. They allow an agent to plan and act optimally in uncertain environments. Although they have been successfully applied to various robotic tasks, they are infamous for their high computational cost. This thesis demonstrates the use of knowledge transfer, learned from previous experiences, to accelerate the learning of POMDP tasks. We propose that in order for an agent to learn to solve these tasks quicker, it must be able to generalise from past behaviours and transfer knowledge, learned from solving multiple tasks, between different circumstances. We present a method for accelerating this learning process by learning the statistics of action choices over the lifetime of an agent, known as action priors. Action priors specify the usefulness of actions in situations and allow us to bias exploration, which in turn improves the performance of the learning process. Using navigation domains, we study the degree to which transferring knowledge between tasks in this way results in a considerable speed up in solution times.

This thesis therefore makes the following contributions. We provide an algorithm for learning action priors from a set of approximately optimal value functions and two approaches with which a prior knowledge over actions can be used in a POMDP context. As such, we show that considerable gains in speed can be achieved in learning subsequent tasks using prior knowledge rather than learning from scratch. Learning with action priors can particularly be useful in reducing the cost of exploration in the early stages of the learning process as the priors can act as mechanism that allows the agent to select more useful actions given particular circumstances. Thus, we demonstrate how the initial losses associated with unguided exploration can be alleviated through the use of action priors which allow for safer exploration. Additionally, we illustrate that action priors can also improve the computation speeds of learning feasible policies in a shorter period of time.

Acknowledgements

This research has taken place at the University of Witwatersrand within the department of Computer Science and Applied Mathematics. This work was funded by the National Research Foundation (NRF) and the Council for Scientific and Industrial Research (CSIR) along side the Department of Science and Technology (DST). The financial assistance of the NRF and DST towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF nor the DST.

I would like to express my sincere gratitude to Benjamin Rosman and Pravesh Ranchod for their supervision on my M.Sc thesis. Their advice and suggestions throughout my studies have been nothing short of valuable. I appreciate their continued support and helpful comments which have largely improved my work. Above all, I am grateful for their time and patience throughout the course of my research as they have always been willing to take time out of their days to ensure I deliver a thesis that is worth while.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Ntokozo Mabena)

Contents

1 Introduction **11**

2 Background **15**
2.1 Introduction . . . 15
2.2 Markov Decision Process . . . 15
2.2.1 Markov Implication in an MDP . . . 15
2.2.2 MDP Formalism . . . 16
2.2.3 Discounting . . . 16
2.2.4 Policies . . . 16
2.2.5 Value Iteration . . . 19
2.2.6 Reinforcement Learning . . . 20
2.2.7 Q-Learning . . . 23
2.3 Partially Observable Markov Decision Process . . . 24
2.3.1 POMDP Formalism . . . 24
2.3.2 History Tracking . . . 25
2.3.3 Policies . . . 27
2.3.4 Value Iteration . . . 31
2.4 Point-Based Partially Observable Markov Decision Process Algorithms 33
2.4.1 Limiting the Size of the Value Function . . . 33
2.4.2 Value Function Updates . . . 34
2.4.3 Policy Execution . . . 35
2.4.4 Value Iteration . . . 36
2.4.5 Initialising the Value Function . . . 36
2.4.6 Core Parameters . . . 37
2.5 Successive Approximations of the Reachable Space under Optimal Policies (SARSOP) . . . 38
2.5.1 Algorithm Overview . . . 39
2.5.2 Sampling . . . 42
2.5.3 Pruning . . . 45
2.6 Action Priors . . . 46
2.6.1 State-based Action Priors . . . 47
2.6.2 Perception-based Action Priors . . . 48
2.7 Conclusion . . . 50

3 Research Method **52**
3.1 Introduction . . . 52
3.2 Research Motivation and Hypothesis . . . 52
3.3 Overview of the Research Method . . . 55
3.3.1 Learning Action Priors . . . 56
3.3.2 SARSOP with Action Priors . . . 62
3.3.3 Experiment Design . . . 69

3.3.4 Experiment Performance Metrics 70
3.4 Conclusion 71

4 Results and Analysis **72**
4.1 Introduction 72
4.2 Experiments 72
4.2.1 Maze Domain 73
4.2.2 Lattice Domain 82
4.2.3 Light-Dark Domain 91
4.2.4 Hallway Domain 95
4.3 Conclusion 99

5 Related Work **101**
5.1 Action Transfer 101
5.2 Shaping Rewards 103
5.3 Options 105
5.4 Policy-Contingent Abstraction (PolCA+) 106
5.5 HQ-Learning 108

6 Future Work **110**

7 Conclusion **112**

References **114**

List of Figures

1 Policy changes with respect to living reward. 18
2 MDP state transitions. 20
3 Value Iteration. 21
4 Synchronous interaction between agent and world. 21
5 POMDP belief state transitions. 27
6 The difference between a true state and belief over states. 27
7 Value function for a two-state problem. 29
8 Belief space and value function for a three-state problem. 29
9 A value function with dominated α-vectors. 30
10 A Graphical representation of the Belief Space $\mathcal{B}$, Reachable Belief Space $\mathcal{R}(b_0)$ and Optimally Reachable Belief Space $\mathcal{R}^*(b_0)$. 39
11 The Formation of Belief Tree $T_{\mathcal{R}}$ Rooted at b_0. 40
12 An example of a lower bound and upper bound on an optimal value function $V^*(b)$. 41
13 Clustering of beliefs based on their initial upper bound and entropy. . . 44
14 δ-Dominance. 46
15 The maze domain. 74
16 Agent's perception capability in maze domain. 75
17 Action prior learning technique results in maze domain. The results are averaged over 16 tasks. 75
18 Threshold experiment for SARSOP with Action Priors for Pruning. The results are averaged over 16 tasks. 77
19 Action prior use approach results. The results are averaged over 16 tasks. 79
20 Correct vs incorrect action priors. The correct action priors are learned from solving multiple tasks, while the incorrect priors are established by inverting the knowledge learned from learning the correct priors. The results are averaged over 16 tasks. 80
21 Maze domain with goal region colour change. 81
22 Incorrect usage of action priors. This involves taking the action priors learned from the domain presented in Figure 15a and using them to solve tasks in the domain presented in Figure 21. The results are averaged over 16 tasks. 82
23 The lattice domain. 83
24 Action prior learning technique results in lattice domain. The results are averaged over 48 tasks. 84
25 Performance results in lattice domain. The results are averaged over 48 tasks. 86
26 Lattice domain location comparison. 87
27 Agent's improved perception capability in lattice domain. 87
28 Action prior learning technique results for precise observations vs ambiguous observations. The results are averaged over 48 tasks. 88

29 Performance results in lattice domain using an improved observation sensor. The results are averaged over 48 tasks. 89
30 Convergence speed results in lattice domain. The results are averaged over 48 tasks. 91
31 Light-Dark domain. 92
32 Stochasticity of action to move North in light-dark domain. 92
33 Examples of trajectories when using action priors. 93
34 Performance results in light-dark domain. The results are averaged over 16 tasks. 94
35 The hallway domain. 96
36 Action prior learning technique results in hallway domain. The results are averaged over 50 tasks. 97
37 Performance results in hallway domain. The results are averaged over 50 tasks. 98
38 An auxiliary and primary task, together with their optimal policies and value functions. 103

List of Algorithms

1 Generalised Point-Based Value Iteration. 36
2 SARSOP. 40
3 α-Vector Backup at a Node b. 41
4 Sampling. 42
5 ε-greedy Q-learning with State-based Action Priors (ε-QSAP). 48
6 ε-greedy Q-learning with Perception-based Action Priors (ε-QPAP). . . 50
7 Learning Perception-based Action Priors in a MDP. 58
8 Learning Perception-based Action Priors in a POMDP. 61
9 SARSOP with Action Priors for Sampling. 65
10 SARSOP with Action Priors for Pruning. 67
11 SARSOP with Action Priors for Simulations. 68

Abbreviations

MDP	Markov Decision Process
COMDP	Completely Observable Markov Decision Process
POMDP	Partially Observable Markov Decision Process
SARSOP	Successive Approximations of the Reachable Space under Optimal Policies
ε-QSAP	ε-greedy Q-learning with State-based Action Priors
ε-QPAP	ε-greedy Q-learning with Perception-based Action Priors
PBVI	Point-Based Value Iteration
ROS	Robot Operating System

Chapter 1

1 Introduction

Autonomous robots that operate in real and complex environments are required to be able to sense the state of the environment to some extent and must be able to take actions that affect the state. However, autonomous robots that operate in these environments are rarely, if ever, able to access the true state of the world or system. Therefore, an essential capability for autonomous robots having to operate reliably in uncertain environments is their ability to handle uncertainty for efficient task planning with imperfect state information. Task planning with imperfect state information refers to how a robot plans to act in order to execute a task without being able to accurately determine the state of the world in which it operates.

Take for example a robot having to navigate from a tunnel entrance to a goal position inside the tunnel. In this case, its task is to navigate from the tunnel entrance to the goal position and its state can refer to its position in space. In order to complete this task, the robot needs to decide on which way to travel so that it reaches its goal as quickly as possible. The process of determining this optimal strategy is known as planning. However, as a result of limited sensing ability and lack of GPS underground, the robot may not be able to confidently determine its location. As a result of the information it is able to gather in this situation, the robot never knows its exact position in space and is therefore forced to perform approximations in order to localise itself. Thus, despite the imperfect information about its state (or position), the robot must determine an exploration strategy that allows it to find the goal. This is a typical example of task planning with imperfect state information.

Subterranean spaces such as mines, tunnels, caves and sewers are abound with factors that foster uncertainty. In such environments, robot performance may be affected by factors including rugged terrain, unanticipated collapses and maze-like tunnels. These factors can be detrimental to robot autonomy. However, it is not only the physicality of these factors that hinders robot capability, but additionally the problem is handling uncertainty.

The likelihood that an autonomous robot will encounter some unforeseen system state during operation is high, as a number of uncertainties may arise from factors, such as sensor noise, missing information, occlusions, imprecise actuators and the absence of GPS in underground environments. All these factors can reduce a robot's ability to correctly interpret its state during task execution.

Partially Observable Markov Decision Processes (POMDPs) provide a principled mathematical framework for planning under uncertainty and allow an agent to reason about the consequences of actions and observations with respect to the agent's limited perception of the environment (Cassandra, Kaelbling, and Littman 1994).

In a POMDP, we maintain a probability distribution over states, known as a belief, and systematically reason over the set of all beliefs, referred to as the belief space. POMDPs consider uncertainty in robot control, sensory information and environment changes, when planning which action is best to take. They attempt to find a feasible policy and to achieve robust performance under such conditions.

Using POMDPs has led to improved performances in various robotic tasks, as a result of incorporating uncertainty into planning. However, due to the complexity associated with planning in the belief space and reasoning over a wide branching factor of different action sequences, POMDPs can be very computationally demanding [Cassandra 1998][Kaelbling et al. 1998][Guy et al. 2013].

The complexity of planning with robots that solve decision problems under uncertainty is compounded as the number of actions available to the robot grows. This is because the complexity of this planning grows exponentially with the number of possible actions. This potentially overshadows the benefits of an agent having a large repertoire of skills, as the added complexity resulting from having a broad skill set may undermine the benefits.

Humans face a similar problem. When a person with access to a rich palette of skills is faced with a task, only a few of these actions could be seen to make sense at that particular moment. Thus, by only considering the useful actions in a situation, it seems likely that humans can successively control the computational explosion of reasoning through chains of actions. By examining ones own thoughts over what has been previously learned in the same or similar circumstance, the useful actions in a particular setting can be determined. This could be a gradual lifelong process, allowing the person to become an expert in situations where he or she is faced with that particular scenario.

The benefits of a developmental and life-long approach can be observed through this example and supports the belief that this is an important component in the development of skills. Therefore, if we allow an agent to be equipped with a mechanism that allows it to determine which actions are most appropriate under particular conditions, perhaps we can improve the computational efficiency of learning new tasks in uncertain environments.

Deciding which actions are useful in a situation can be determined by using past experiences of optimal behaviours from multiple previously solved tasks [Rosman and Ramamoorthy 2012]. Because there is some underlying structure common to all these learned tasks as they exist in the same or similar environment, it is this structure we hope to take advantage of to facilitate faster learning. Hence, we are concerned with allowing agents to be able to learn domain invariances which act as a common sense knowledge of the domain in which it operates. By forming better abstractions of the domain and in turn learning domain invariances that provide insights to elements that are common to a large class of behaviours, this knowledge can offer an understanding in learning which behaviours should be avoided in particular situations. As such, instead of taking into account a form of model learning that considers the reward structure for any individual task, we are interested in a form of model learning that considers the

commonalities between a set of tasks.

We propose that for an agent to be generally capable when it is required to perform a range of unknown tasks in which it experiences uncertainty, the agent must have the ability to continually learn from a lifetime of experience. This is largely dependent on its ability to generalise from past experiences and form representations which facilitate faster learning and transfer of knowledge between different situations. By exploiting the commonalities between large families of tasks, the agent may potentially minimise situations where it has to relearn from scratch, as well as build better models of the domain. Therefore, we hope to facilitate faster learning by learning such regularities from the domain and extracting the common elements between tasks.

We propose a method that extends existing POMDP formalisms by taking into account the statistics of action choices over a lifetime of the agent. This is an effort to allow the agent to be able to quickly cut down options when deciding which actions to select, in a manner which may not have been obvious if each task was solved in isolation. We thus show how the agent can reuse behavioural knowledge learned from previously solved tasks to accelerate the learning of new tasks.

In this work, our goal is to show how an agent acting in a setting where it is uncertain about the current state of the world or system can benefit through the use of having a prior knowledge over actions indicating the preference of the agent in taking particular actions in certain situations. We demonstrate these advantages in navigation domain experiments, where we show that using an agent's experience of past behaviours can lead to convergence speed improvements.

The rest of this thesis is structured as follows. In Chapter 2, we begin with a thorough background on Markov Decision Processes (MDPs), in Section 2.2. The reinforcement learning paradigm is also explained in this section in order to give an understanding of the techniques that will be used to gather and conceptualise action priors in a case where the state space is fully observable. Thereafter, we move on to give a detailed background on POMDPs, the belief space MDP, in Section 2.3. We then give a general discussion on point-based POMDP algorithms, in Section 2.4, as an introduction to the detailed description of the Successive Approximations of the Reachable Space under Optimal Policies (SARSOP) algorithm presented in Section 2.5. Following Section 2.5, we present the idea of action priors and discuss the theory behind how they can be learned in Section 2.6. Then, in Chapter 3, we discuss the methods and techniques of how we aim to achieve the goals of this research by, firstly, presenting the motivation behind the research and, secondly, defining our research hypothesis, which can both be seen in Section 3.2. We then follow to give an overview of our research methodology, in Section 3.3, describing how we intend to validate our hypothesis, and explain our experimental procedure with regards to learning action priors and discuss three proposed algorithms that we believe will display the benefits of using action prior knowledge in solving POMDP tasks. In chapter 4, we present the details of our experiments and illustrate the results obtained from carrying out these experiments by presenting a summary of the experiments' results. We also provide a detailed analysis of the produced experiment results along with each experiment, to give an understanding

of the obtained and presented data. Finally, we end off by giving related research work produced by other authors in Chapter 5, the relative directions that future research, related to our research work, may take (see Chapter 6) and present our concluding remarks in Chapter 7.

Chapter 2

2 Background

2.1 Introduction

In this chapter, we introduce the background material for understanding the SARSOP algorithm and the concept of action priors. Firstly, we present a detailed description of the MDP framework, in Section 2.2, so as to give an understanding behind reinforcement learning practices, which form an essential underpinning for how action priors will be gathered in a case where the state space is fully observable. Following the MDP framework, we present the framework of POMDPs, in Section 2.3, as these are two useful frameworks in the realm of decision-making. We then describe the transition from MDPs to POMDPs, along with a discussion of the algorithms that are entailed under each paradigm. In Section 2.4, we discuss the details of point-based POMDP methods and how these algorithms are structured in order to compute a policy. We then move on to explain the workings of the SARSOP algorithm in Section 2.5. Thereafter, we give an overview of the ideas behind action priors and the gathering of prior knowledge over actions in Section 2.6.

2.2 Markov Decision Process

2.2.1 Markov Implication in an MDP

"Markov" refers to the Markov assumption which implies that given a particular state, the history and future from that state are independent from each other. For an MDP, "Markov" means the action outcomes of a state depend only on that current state and the selected action [Sutton and Barto 1998]. This implies that given a sequence of occurrences and their associated time indices, if the agent is able to determine the current state in which it lies, then what may have occurred in the past is not required to determine what may occur in future. In general:

$$P\left(s_{t+1}=s', r_{t+1}=r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\right) = P(s_{t+1}=s', r_{t+1}=r | s_t, a_t) \tag{1}$$

where $s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0$ are all possible values of past events [Sutton and Barto 1998]. This implies a policy search process where the successor function only depends on the current state.

2.2.2 MDP Formalism

Markov Decision Processes (MDPs) provide a mathematical framework for modeling decision making problems. They are the basis for solving probabilistic task problems in decision making and can be used to model stochastic or deterministic search problems [Puterman 1994][Mitchell 1997]. When we consider MDPs for planning, the underlying idea refers to how an agent can optimally *plan* in order to discover the best action-selection strategy defining how the agent should behave. [Klein and Abbeel 2013]. Therefore, the goal of the MDP is to model a task and find the best policy, or action sequence that enables an agent to solve its tasks.

An MDP is defined as a tuple (S, A, T, R, γ), where S is a finite set of *states*, A is a finite set of *actions* that the agent can take, $T : S \times A \times S \rightarrow [0, 1]$ is a *transition function* which gives the probability of transitioning from one state to another when taking a particular action, $R : S \times A \times S \rightarrow \mathbb{R}$ is a *reward function* that indicates the immediate payoff of taking a particular action at a given state and $0 \leq \gamma \leq 1$ is the discount factor. In particular $T(s, a, s')$ gives the probability of transitioning from state s to state s' after taking action a (i.e. $P(s'|s, a)$) and $R(s, a, s')$ is the reward received by the agent when transitioning from state s to state s' as a result of taking a. Because $T(s, a, s')$ is a probability function, $\sum_{s' \in S} T(s, a, s') = 1, \forall a \in A, \forall s \in S$.

2.2.3 Discounting

Because we consider sequences of rewards rather than a single utility at the end of an episode, an agent - through some manner - should be able to optimise the sequence of rewards that it discovers. The idea is that when there is a level of uncertainty, an agent's preference is to receive cumulative rewards as it moves from one state to another rather than receiving a single large reward equal to the sum of the rewards received during step-wise reward accumulation [Klein and Abbeel 2013]. The consequence where an agent prefers to receive rewards earlier rather than later, as a result of earlier rewards yielding a higher utility, introduces the concept of *discounting* [Mitchell 1997]. Discounting is also useful in ensuring convergence. It allows rewards received earlier to yield a higher utility than those received later. Without discounting, the agent could potentially find a way to keep receiving rewards indefinitely.

2.2.4 Policies

The action-selection strategy found in MDP planning is called a *policy* $\pi : S \rightarrow A$ where $\pi(s)$ gives an action for each state. Put simply, a policy is a mapping from states to actions. In the case of deterministic single-agent search problems, the objective is to find an optimal *plan* (or sequence of actions) from a start state to a goal state [Sutton and Barto 1998]. In MDPs, an agent attempts to find an *optimal policy* $\pi^* : argmax_\pi U^\pi$ where U^π is the *utility* or *return.* The utility U^π is the accumulated discounted reward

received from following the policy π and is defined to be

$$U^{\pi} = \sum_{t} \gamma^{t} r_{t}, \tag{2}$$

where r_t is the reward at time t [Mitchell 1997]. This is known as the value of a policy. An optimal policy π^* is a policy that maximizes expected utility when followed [Klein and Abbeel 2013]. Thus, the goal of a learning agent is to find such a policy.

We can also define the value of a policy π that starts at state s by computing the expected rewards for starting in state s and following that policy . The value $V^{\pi}(s)$ of starting in state s and executing policy π can be defined to be

$$V^{\pi}(s) = \sum_{s' \in S} T(s, \pi(s), s') \left[R(s, \pi(s), s') + \gamma V^{\pi}(s') \right]. \tag{3}$$

Note that where there exists a terminal state, its value will always be zero.

The value function $V^{\pi} : S \to R$ gives the expected discounted sum of rewards for executing π starting at state s (i.e. V^{π} is value function for an arbitrary policy π). Thus, given a value function $V^{\pi}(s)$, we may define a policy π to be

$$\pi(s) = \arg\max_{a} \sum_{s' \in S} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi}(s') \right]. \tag{4}$$

Similarly, given a policy π, we define an action-value function $Q^{\pi}(s, a)$ as

$$Q^{\pi}(s, a) = \sum_{s' \in S} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi}(s') \right] \tag{5}$$

which gives the expected utility of taking action a in state s and thereafter acting according to π. We refer to Q^{π} as the action-value function for policy π.

By maintaining an average, for each state encountered, of the actual returns that have followed from that state when an agent follows a policy π, the average will converge to the state's value $V^{\pi}(s)$ as the number of times that state is encountered approaches infinity. Similarly, if separate averages are kept for each action taken in a state, then these averages will converge to the action values $Q^{\pi}(s, a)$ as the number of times that state is encountered approaches infinity. Therefore, the value functions V^{π} and Q^{π} can be estimated from experience.

Because the agent's action outcomes are potentially non-deterministic, we cannot restrict it to computing a particular sequence of actions without taking into account every probable state. The complication is that many relevant state-action pairs may never be visited. This is a detrimental issue because the purpose of learning action values is primarily to help in selecting the best action to take among all the possible actions in each state. In order for an agent to efficiently compare alternatives between actions, it must be able to compare the value of all the actions from each state, not just the one it just so happened to experience or one we currently favour. The agent should precisely know the best action to take for every possible state.

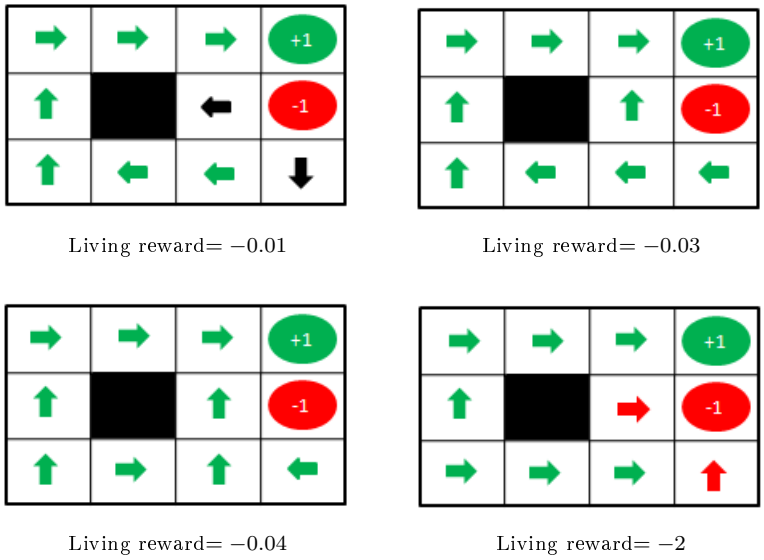


Figure 1: Policy changes with respect to living reward.

In a setting such as an MDP, it is possible to have more than one optimal policy. This is possible because an agent may have a number of different policies which maximize its expected sum of rewards [Klein and Abbeel 2013].

We can also observe that different policies may be considered to be optimal depending on the manner in which the reward function is specified. Because the reward function defines what the good and bad events are for the agent, it may serve as a basis for altering the policy. Given the same domain, an optimal policy may change as a result of changes in the reward function. This idea is depicted in Figure 1.

Figure 1 illustrates the policy changes as a result of reward function changes for an agent having to navigate to its goal location. The green oval in the figure represents a goal location, while the red oval represents a danger state. The reward function is as follows. The reward for reaching the goal state is 1, the reward for reaching the danger state is -1, while the reward for transitioning from one state to every other state other than the goal and danger state is specified by the living reward. The arrows in the figure show the optimal action for an agent to take in a particular state given the relative reward function. The green arrows represent actions that lead to the goal state (useful actions), the red arrows represent actions that lead to the danger zone (dangerous actions), while the black arrows represent actions that do not lead the agent anywhere (useless actions).

There are three quantities that are of interest in obtaining a solution to an MDP, namely a policy π, a value function V and the action-value function $Q(s, a)$ (See Figure 2 for an intuition of an action-value). When we consider two policies , policy π and π' , we define π to be better than or equal to π' if its expected return is greater or equal to that of π' for all states. More formally, $\pi \geq \pi'$ if and only if $V^{\pi}(s) \geq V^{\pi'}(s), \forall s \in S$

where s is a state and S is the set of all states. An optimal policy π^* defined as

$$\pi^*(s) = \arg\max_{\pi} U^{\pi} \tag{6}$$

is a policy that is better than or equal to all other policies. There exists at least one such policy in any MDP problem. In cases where there exists more than one optimal policy π^*, they share the same optimal value function V^* defined as

$$V^*(s) = \max_{\pi} V^{\pi}(s) = \sum_{s' \in S} T(s, \pi^*(s), s') \left[R(s, \pi^*(s), s') + \gamma V^*(s') \right] \tag{7}$$

where $V^*(s)$ is the expected utility of starting in state s and following an optimal policy π^* [Mitchell 1997]. Optimal policies also share the same optimal action-value function Q^*defined as

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) = \sum_{s' \in S} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s') \right] \tag{8}$$

where $Q^*(s, a)$ gives the expected utility of having taken action a in state s and thereafter acting optimally [Klein and Abbeel 2013].

We can rewrite the optimal policy π^* in terms of V^* as

$$\pi^*(s) = \arg\max_{a} \sum_{s' \in S} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s') \right] \tag{9}$$

which is essentially just a greedy policy with respect to V^*. On the other hand, V^* can be written in a special form without reference to any specific policy as

$$V^*(s) = \max_{a} \sum_{s' \in S} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s') \right]. \tag{10}$$

This is known as the Bellman equation for V^*. The Bellman optimality equation for Q^* is

$$Q^*(s, a) = \sum_{s' \in S} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s') \right]. \tag{11}$$

The optimal sought after quantities in an MDP are therefore the optimal value (utility) of a state $V^*(s)$, the optimal value (utility) of a Q-state $Q^*(s, a)$ and the optimal policy $\pi^*(s)$.

2.2.5 Value Iteration

One way of understanding value iteration is by reference to the Bellman optimality equation (see Equation 10). Although the Bellman equation gives us a way of getting at V, it is recursive, and so one approach to solving this is through iteration. By turning the Bellman optimality equation into an update rule, we obtain the value iteration method for solving V.

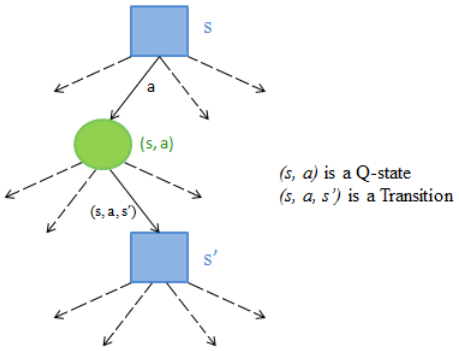


Figure 2: MDP state transitions.

The value of a state at time t is computed to be

$$V_{t+1}(s) \leftarrow \max_a \sum_{s' \in S} T(s, a, s') \left[R(s, a, s') + \gamma V_t(s') \right] \tag{12}$$

where $V_0(s) = 0, \forall s \in S$ [Sutton and Barto 1998]. Equation 12 is iterated until convergence is reached. In practice, convergence is reached when the value function changes by only a small amount between iterations. However, for value iteration to converge exactly to V^*, it needs an infinite number of iterations. Thus, $\lim_{t \to \infty} V_{t+1}(s) = V^*$.

With value iteration, every iteration updates the current values by a very small amount. Therefore, convergence is guaranteed because for value iteration that iterates sufficiently often, the difference between the updated and previous values will ultimately be equal to 0. This result is due to the fact that $\lim_{t \to \infty} \gamma^t = 0$ [Mitchell 1997].

Value iteration converges to unique optimal values as at each time step the value approximations are recomputed to values closer to those of the optimal quantities [Mitchell 1997]. It updates both, the values and the policy. Although we do not track the policy as the algorithm iterates, taking the maximum over executable actions implicitly recomputes it.

2.2.6 Reinforcement Learning

When we think about the nature of behaviour learning, the idea that an agent learns through its interaction with its environment is most likely the first to occur us . Take new born babies for example. As they learn how to perform tasks such as crawling, walking or playing, they have no explicit teacher, but they do have a direct connection with their environment as a result of their sensory and motor functions (see Figure 4). Continuously exercising this connection results in a rich source of information over the consequences of actions and how to go about achieving particular goals. Throughout our lives, it is these interactions that play a primary role in gaining the accrued knowledge about ourselves and the environment in which we live. With everything we do, whether

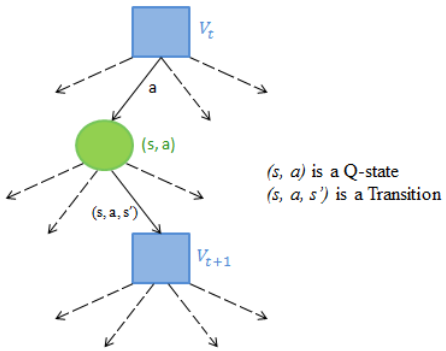


Figure 3: Value Iteration.

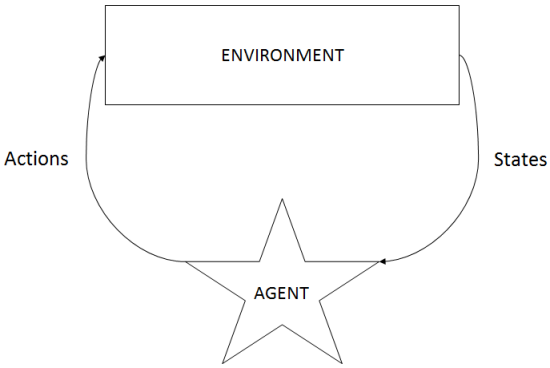


Figure 4: Synchronous interaction between agent and world.

we are learning how to play an instrument or how to behave in a classroom, we are constantly alert of how our environment responds to our actions.

Reinforcement learning is a computational approach to learning from interaction. It is a formalism for finding a solution to the problem of not knowing how to act in an unknown system or environment and is concerned with learning how to map situations to actions [Sutton and Barto 1998].

Under the reinforcement learning paradigm, it typically requires that the problem is modeled as an MDP. Thus, the reinforcement learning problem involves solving the MDP. However, the transition model $T(s, a, s')$ and the reward function $R(s, a, s')$, discussed in Section 2.2.1 and 2.2.3, in this case is typically unknown [Klein and Abbeel 2013]. Here, an agent is encouraged to act in its environment and through its actions, the agent can then learn about the system which will, in turn, allow the agent to perform planning. The idea is that in reality, the transition model and reward function are not known before these are learned through experience.

The key ideas in reinforcement learning are *exploration*, *exploitation*, *regret* and *sampling*. Exploration refers to taking actions that are not recommended by the current estimate of the model so as to acquire more information (or experience). This allows

the learning agent to explore the environment, receive feedback for taking different actions and consequently learn new knowledge from the environment in which it acts. Exploitation refers to taking the best action according to the current estimate of the model in order to maximise the expected reward. Exploration and exploitation are complementary, but opposite. Exploration leads to the maximisation of the expected reward in the long run at the risk of losing short term reward, while exploitation maximises the short term expected reward at the price of losing the gain in the long run.

For example, suppose the value of the greedy action is known with certainty, in comparison to several other actions that are estimated to be nearly as close to that of the greedy action with substantial uncertainty. As a result of the uncertainty, it could be that one of the other actions actually has a higher value than that of the greedy action. However, this action would not be able to be determined. Thus, it may be more fruitful to explore the other actions and discover which one of them have a higher value than taking the greedy action. Although the reward is lower in the short run, during exploration, the reward may be higher in the long run because after discovering the better actions you would then be able to exploit them rather than taking the greedy action. Therefore, there exists a conflict between exploration and exploitation since it is not possible both to explore and exploit, simultaneously, using a single action selection.

Sampling refers to probabilistically selecting (possibly suboptimal) actions from some distribution during the course of exploration. Regret on the other hand refers to the expected decrease in reward when an agent's performance is compared to that of an agent which acts optimally from the very beginning. In other words, regret is a measure of how much worse the agent performed than if it had known the best strategy before hand. In order for an agent to act optimally, it must be able to reason about the long term consequences of its actions.

Reinforcement learning refers to learning how to behave from interaction in order to achieve a goal. The underlying structure of reinforcement learning is as follows. If the environment satisfies the Markov property (see Section 2.2.1), then the environment is therefore an MDP and the model consists of the one step transition probabilities given by $T(s, a, s')$ and expected rewards for all states and their possible actions. All learning is based on observed samples of outcomes and the agent must learn to act so as to maximise expected rewards. Therefore, the goal of the agent is to learn the optimal policy $\pi^*(s)$ so as to maximise the return of future rewards. The agent receives feedback from interacting with the environment in the form of rewards which are specified by the reward function $R(s, a, s')$.

Reinforcement learning is a computational approach that allows us to understand and automate goal directed learning and decision making. Its methods/algorithms are typically divided into two classes: model-based and model-free approaches.

Model-Based Learning

The goal in model-based learning is to learn the transition model $T(s, a, s')$ and reward function $R(s, a, s')$. An agent must learn an approximate model based on its experiences in order for it to be able to learn a policy which it can compute if the transition model and reward function are known [Klein and Abbeel 2013]. In other words, the agent must accumulate experiences, build a transition and reward model based on those experiences and then solve for the policy with the assumption that the model is correct. This is achieved by

1. Learning the empirical model: Taking actions, either randomly or using a more efficient strategy, to observe the rewards received by transitioning from one state to another and then building an estimate of the transition model $T(s, a, s')$ and reward model $R(s, a, s')$.

2. Solving the learned MDP: Using the empirical model to do planning so as to find a policy π.

Thus, there are two notable phases under model-based learning. These correspond to collecting experience in order to build a transition model and using that model to do planning in the hope that the action selection strategy from planning is more effective than blindly attempting actions.

Model-Free Learning

Model-free learning involves the direct finding of a policy without any explicit knowledge of the dynamics of the environment or the consequences of actions. The only information it requires is of the states that exist and the possible actions that may be executed in each state. It differs from model-based learning in that an agent is able to solve for values $V(s)$ and $Q(s, a)$ by simply gathering experience in the MDP environment, rather than first learning the transition model $T(s, a, s')$ and reward function $R(s, a, s')$ [Klein and Abbeel 2013]. The goal is to directly estimate the values of V^*, Q^* and π^* without establishing a model.

2.2.7 Q-Learning

One commonly used iterative method for computing Q-values is *Q-learning* [Sutton and Barto 1998]. It is a model-free reinforcement learning method whereby the learned action-value function Q directly approximates the optimal action-value function Q^*. A Q-value at time t is updated using

$$Q_{t+1}(s, a) \leftarrow (1 - \alpha)\, Q_t(s, a) + \alpha \left(R(s, a, s') + \gamma \max_{a'} Q_t(s', a') \right) \tag{13}$$

known as the *Q-value update* where $\alpha \in (0, 1]$ is the learning rate [Klein and Abbeel 2013]. The core of the algorithm is a value iteration update with the Q-function $Q(s, a)$ iteratively converging to the optimal Q-function Q^*.

This approach is referred to as *off-policy learning* [Sutton and Barto 1998]. Off-policy learning allows an agent to be able to learn about the optimal policy π^* without ever having to follow the optimal policy, by gathering information from random action selections.

2.3 Partially Observable Markov Decision Process

2.3.1 POMDP Formalism

With MDPs we assume that the agent has full observability of the environment. This means after executing a particular action whose outcome is not known in advance, the agent is able to determine without any ambiguity the resulting state after that action is taken. However, many real world problems do not fit the assumptions of the MDP framework. This is because an agent acting in a real world environment may not be able to directly observe the state of the environment at every time step.

Consider a robot having to navigate through an environment using an array of sensors. In this case, robotic sensors such as cameras and lasers can only provide limited/partial information about the environment because the sensors are unable to give full knowledge of the domain. For example, the robot sensors cannot provide information beyond the range of the sensors and cannot directly observe the contents of the opposite side of a wall/obstacle. For this reason, these environments are referred to as partially observable domains.

POMDPs provide a principled mathematical framework for planning in uncertain and dynamic environments and allow us to model sequential decision making problems under uncertainty [Sondik 1971][Kaelbling 1998][Png 2011]. They allow an agent to be able to compute a policy in cases where it is not able to directly determine the state in which it currently lies [Kaelbling 1998]. A POMDP is thus an extension and generalisation of an MDP to circumstances in which states are not fully observable. In order for an agent to act optimally in an MDP setting, it only needs to consider the current state in which it lies, while in a POMDP setting an agent may need to take into account the previous history of all observations and actions [Braziunas 2003].

A POMDP is defined by a tuple $(S, A, \mathcal{O}, T, Z, R, \gamma)$ where S is a finite set of *states*, A is a finite set of *actions* that the agent can take, $\mathcal{O}$ is a finite set of *observations*, $T : S \times A \times S \to [0, 1]$ is a *transition function* which gives the probability of transitioning from one state to another when taking a particular action, $Z : S \times A \times \mathcal{O} \to [0, 1]$ is an *observation function* which gives the probability of observing an observation in a some state after taking a particular action, $R : S \times A \times S \to \mathbb{R}$ is a *reward function* that indicates the immediate payoff of taking a particular action at a given state and $0 \leq \gamma \leq 1$ is the discount factor. In particular $T(s, a, s')$ gives the probability of transitioning from state s to state s' after taking action a (i.e. $P(s'|s, a)$), $Z(s, a, o)$ gives the probability of observing observation o after taking action a in state s (i.e. $P(o|s, a)$) and $R(s, a, s')$ is the reward received by the agent when transitioning from state s to state s' as a result of taking action a. Because $T(s, a, s')$ and $Z(s, a, o)$ are

a probability functions, $\sum_{s' \in S} T(s, a, s') = 1, \forall a \in A, \forall s \in S$ and $\sum_{o \in \mathcal{O}} Z(s, a, o) = 1, \forall a \in A, \forall s \in S$. We note that a POMDP is an extension of the MDP paradigm with the addition of an observation space $\mathcal{O}$ and observation function $Z(s, a, o)$.

The goal of a learning agent in this paradigm is to maximise the total expected reward when taking a sequence of actions in a domain with state uncertainty.

2.3.2 History Tracking

Because an agent's immediate observations do not provide it with sufficient information in allowing it to disambiguate its position in space and perform the optimal action, there is a necessity for some form of memory that summarises its previous experiences on which it can correctly base its decisions. This is particularly useful because an agent may not be able to localise itself by making observations of its surroundings, but if it can combine these observations with its staring point along with the actions that it took subsequently, it is likely to have a much better idea of its location.

This memory is referred to as the agent's *history*. A history refers to everything that occurred while a task was being executed. It is comprised of all the agent's interactions with the environment as it performs its relative tasks. In a POMDP setting, a system history starting at time 0 and ending at time t is a sequence of state, action and observation triples [Braziunas 2003]. A history is typically denoted as

$$\langle s_0, a_0, o_0 \rangle, \langle s_1, a_1, o_1 \rangle, \ldots, \langle s_t, a_t, o_t \rangle .$$

Since the system is only partially observable, the agent does not have access to the true state of the world. Therefore, an agent can only make decisions based on its observable history. Assume that the agent has prior beliefs about the world that are summarised by the probability distribution b_0 from the beginning. If the agent takes some action a_0 based only on b_0, the observable history starting at time 0 and ending at time t will therefore be a sequence of action and observation pairs and is denoted as

$$\langle a_0, o_1 \rangle, \langle a_1, o_2 \rangle, \ldots, \langle a_{t-1}, o_t \rangle .$$

However, maintaining and working with extensive histories can quickly become cumbersome and intractable. Instead, a probability distribution over states, known as a *belief*, is maintained to summarise an agent's past experience. This is a sufficient statistic for representing the agent's past history. By basing POMDP policies on beliefs we are able to maintain the Markovian property for the policy as the next belief state depends only on the current belief state and the immediate transition taken (i.e. the action and observation received). As a result, this belief process is a Markov process.

A belief state $b \in \mathcal{B}$ gives the probability of being at each state (See Figure 6), while a belief space $\mathcal{B}$ refers to the set of all possible beliefs. We consider a belief state b as a vector of probabilities and the sum of all these probabilities must sum to 1 i.e. $\sum_{s \in S} b(s) = 1, b(s) \in [0, 1], \forall s \in S,$.

When an agent takes action a at belief state b and observes an observation o, the belief can be updated using

$$
\begin{aligned}
b' = b^{a,o}(s') &= P(s'|b,a,o) && (14)\\
&= \frac{P(s',b,a,o)}{P(b,a,o)} && (15)\\
&= \frac{P(o|s',b,a)P(s'|b,a)P(b,a)}{P(o|b,a)P(b,a)} && (16)\\
&= \frac{P(o|s',a)\sum_{s\in S}P(s'|b,a,s)P(s|b,a)P(b,a)}{P(o|b,a)P(b,a)} && (17)\\
&= \frac{Z(s',a,o)\sum_{s\in S}P(s'|b,a,s)P(s|b,a)}{P(o|b,a)} && (18)\\
&= \frac{Z(s',a,o)\sum_{s\in S}T(s,a,s')b(s)}{P(o|b,a)} && (19)
\end{aligned}
$$

known as the *belief update*, where

$$P(o|b,a) = \sum_{s'\in S} Z(s',a,o) \sum_{s\in S} b(s)T(s,a,s') \tag{20}$$

can be treated as a normalising factor, independent of s' allowing $\sum_{s'\in S} b^{a,o}$ to sum to 1. In this case we can represent the belief update formula as

$$b^{a,o}(s') = \eta Z(s',a,o)\sum_{s\in S} T(s,a,s')b(s) \tag{21}$$

where $\eta = \frac{1}{P(o|b,a)}$ is the normalising constant and $P(o|b,a)$ is from obtained Equation 20. This shows that a POMDP can be seen as a belief MDP, with the MDP defined on a continuous state space, since there are infinite beliefs for any given POMDP. Figure 5 displays a graphical representation of how transitions between belief states are completed.

In Figure 6 we show the difference between the true states and the belief over states of an agent following a sequence of actions in a domain in which it is allowed to navigate. In Figure 6a, the agent is fully capable of precisely determining its state and position. Therefore, because it is certain of the state in which it lies, the probability of being in that state will always be equal to 1.

The sequence of actions in Figure 6 are as follows. If the agent's initial position is that given by 6a)i) and takes an action to move east, it will know with certainty that it lies in the state depicted by 6a)ii). If from 6a)ii) the agent takes an action to move east again, it will know with certainty that it lies in the state depicted by 6a)iii). On the other hand, to illustrate an agent's belief over states we assume that the agent is initially equally likely to be in any of the three states other than its goal location (See Figure 6b)i)). If the agent takes an action to move east from its initial belief and does

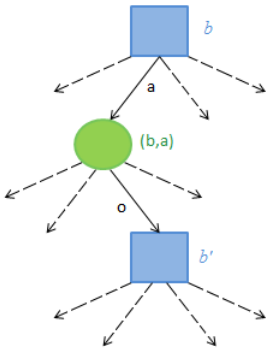
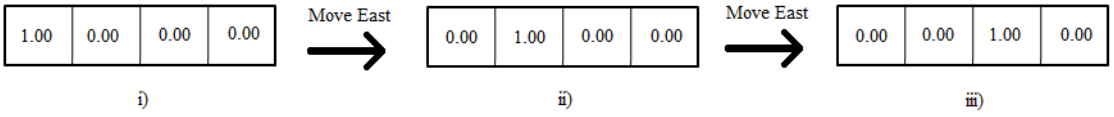
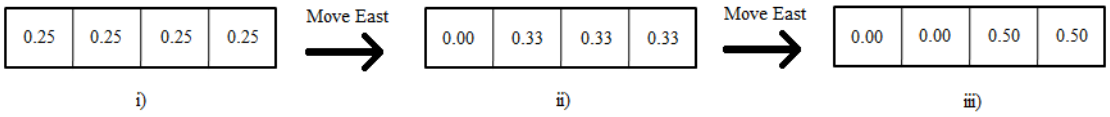


Figure 5: POMDP belief state transitions.



a) True state.



b) Belief over states.

Figure 6: The difference between a true state and belief over states.

not collide with a wall obstacle, then its new belief over states will be that portrayed by 6b)ii). If the agent takes an action to again move east and does not collide with a wall obstacle, then its new belief over states will be that depicted by 6b)iii).

2.3.3 Policies

As in a standard MDP, the goal of a learning agent in a POMDP is to find an optimal policy $\pi^* : \mathcal{B} \rightarrow A$ where $\pi^*(b)$ maps a belief b to a prescribed action a. The major difference between a POMDP policy and an MDP policy is that a POMDP policy maps belief states (which is a probability distribution over states) to actions, while an MDP policy maps states to actions. In a POMDP, an agent is required to determine the value of executing a particular action a from some belief state b in order to learn how to act optimally. It achieves this by maintaining a value function $V : \mathcal{B} \rightarrow \mathbb{R}$ which gives an expected value over world states for executing each action. The expected reward for

following policy π starting at an initial belief state b_0 is defined as

$$V^{\pi}(b) = E^{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t | b_0\right] \tag{22}$$

where r_t is the reward received at time t [Png 2011].

In POMDPs, an agent attempts to find an optimal policy π^* that maximises the expected total reward. By optimizing the long-term reward an agent is able to compute an optimal policy using

$$\pi^* = \arg\max_{\pi} V^{\pi}(b) \tag{23}$$

which yields the policy which returns the highest expected reward for a belief state. The optimal policy π^* can be represented by the optimal value function V^* and as with the MDP model, we can define the Bellman update operator for the belief space MDP as

$$V^*(b) = \max_a \left[\sum_{s \in S} b(s)R(s,a) + \gamma \sum_{o \in \mathcal{O}} P(o|b,a)V^*(\tau(b,a,o))\right] \tag{24}$$

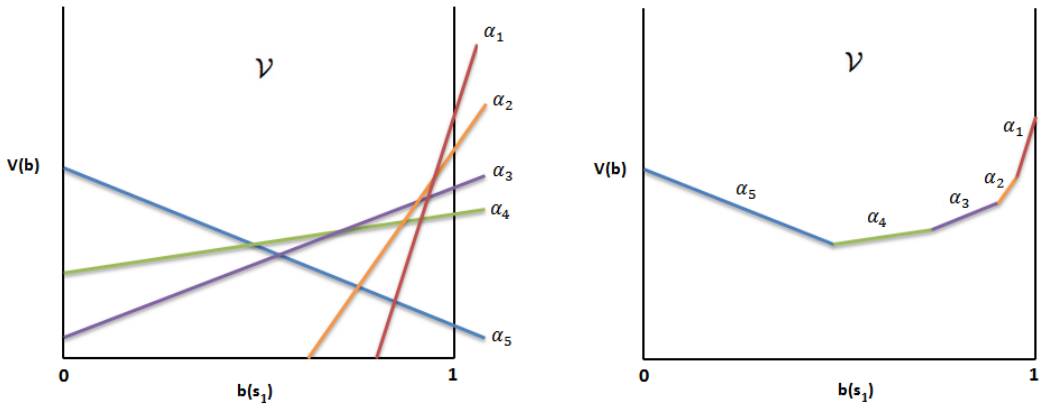
where $\tau(b,a,o) = b'$ is the resulting belief state after taking action a and observing o from belief b. By extracting the policy from the computed optimal value function, an agent is therefore able to discover the optimal policy i.e. $\pi^* = \arg\max_{\pi} V^{\pi}(b)$.

Using a finite set of linear functions, known as α-vectors, the value function for a POMDP can be modeled arbitrarily closely as the upper surface of the α-vectors. Hence we can define the value function over the full belief space as $V = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. Each α-vector is associated with an action and denotes the immediate reward following a given action. Formally, an α-vector has $|S|$ components and can be denoted as $\alpha = \langle V(s_1), V(s_2), \ldots, V(s_n)\rangle$ [Kaelbling 1998]. Thus, using this representation, the optimal value function $V^*(b)$ can be computed by a piecewise-linear, convex function

$$V^*(b) = \max_{\alpha \in \Gamma}(\alpha \cdot b) \tag{25}$$

where Γ is a finite set of vectors called α-vectors, b is a vector representing a belief and $\alpha \cdot b$ is the dot product of the α-vector and belief vector b i.e. $\alpha \cdot b = \sum_{s \in S} b(s) \cdot \alpha(s)$ [Sondik 1971].

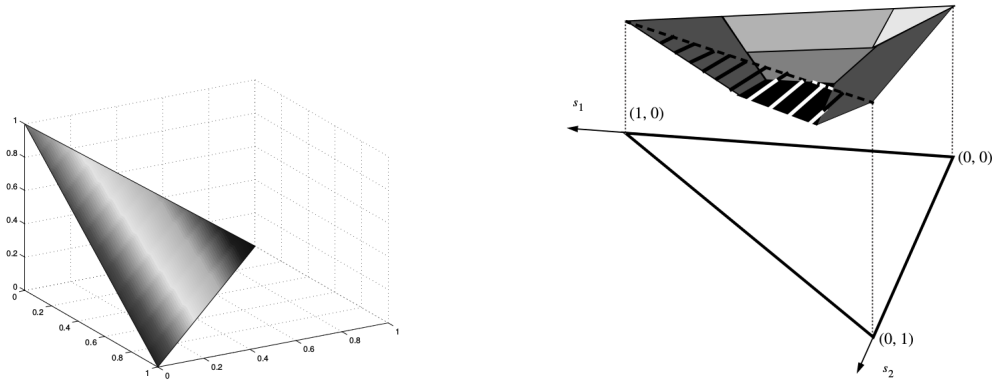
To illustrate the latter, suppose we consider an agent acting in a world with only two states. The belief state vector representation of this world will consist of only two non-negative numbers that sum to 1 i.e. $\langle P(s_1|b), P(s_2|b)\rangle$ or $\langle b(s_1), b(s_2)\rangle$. As a result, a single value between 0 and 1 is sufficient to describe a belief state because with a two state POMDP, if the probability for being in one of the states is known to be p, then it is known that the probability of being in the other state must be $1 - p$. The values of taking particular actions over this belief space can be represented using linear segments and can be seen in Figure 7a. Therefore, the optimal value function over the belief space can be extracted from Figure 7a by taking the maximum of all the line segments at each point in the belief space as illustrated with Figure 7b. The result is the upper



a) The value function for a sample two-state problem.

b) The optimal value function for the sample two-state problem.

Figure 7: Value function for a two-state problem.



a) Belief space for a three-state problem [Roy et al. 2005].

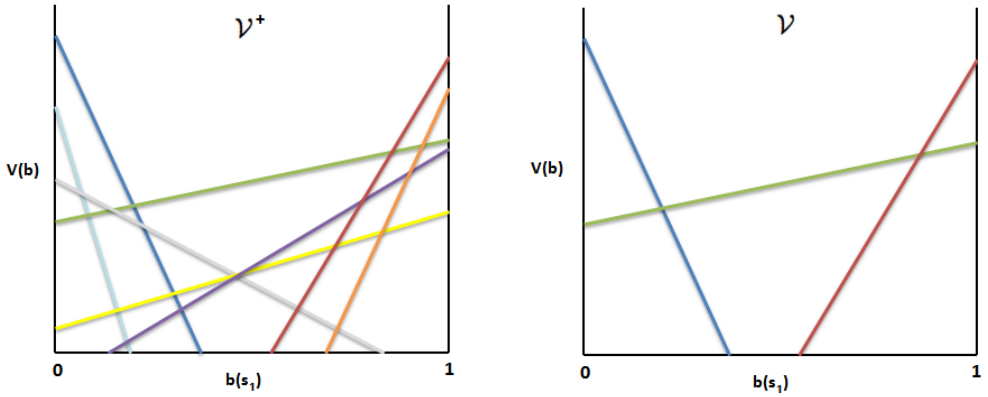
b) A sample value function of a three-state problem [Kaelbling et al. 1998].

Figure 8: Belief space and value function for a three-state problem.

surface of the graph presented in Figure 7a, which gives us the optimal value function over the belief space. We note that a POMDP value function has a piecewise linear and convex property [Sondik 1971][Smallwood and Sondik 1973][Cassandra 1998].

When we consider an agent acting in a world with only three states, then a belief can be determined using only two values. The belief space can be seen to be a two-dimension triangle, while the value function associated with taking particular actions over this belief space is a plane in the three-dimensional space. The optimal value function is a bowl shape that is composed of planar facets [Kaelbling 1998]. These value function characteristics can be observed in Figure 8. In general, a belief state can be represented by $|S| - 1$ values [Cassandra 1998].

An interpretation of Figure 8b may be as follows. The convexity of the optimal value function tells us that if an agent's belief is equal to those beliefs that lie towards



a) Value function with useless α-vectors. b) Parsimonious representation of the value function.

Figure 9: A value function with dominated α-vectors.

the centre of the belief space, then the agent is very uncertain about the true state of the world. In this case, the agent may want to take an action that is optimal given its uncertainty (which may mean the action that is suitable in the widest range of cases). On the contrary, the belief states that lie towards the corners of the belief space suggest that the agent is more certain over the state in which it lies and is therefore able to act more appropriately to a specific situation.

Any value function can be represented as a finite set $\mathcal{V}^+$ of α-vectors. Because many α-vectors in $\mathcal{V}^+$ may be dominated by other α-vectors, as seen in Figure 9, we can use a parsimonious set $\mathcal{V}$, which is a unique minimal subset i.e. $\mathcal{V} \subseteq \mathcal{V}^+$, to represent the same value function. This process of establishing $\mathcal{V}$ from $\mathcal{V}^+$ is called pruning. We refer to an α-vector as not dominated if it is included in this set $\mathcal{V}$. If we consider a set of not dominated α-vectors $\mathcal{V}$ and a particular α-vector, α_1, in $\mathcal{V}$, we define the region of belief space over which α_1 dominates to be

$$\mathcal{R}(\alpha_1, \mathcal{V}) = \{b \,|\, \alpha_1 \cdot b > \tilde{\alpha} \cdot b, \forall b \in \mathcal{B}, \forall \tilde{\alpha} \in \mathcal{V} - \alpha_1\}\,. \tag{26}$$

The simplest pruning strategy is to test $\mathcal{R}(\alpha, \mathcal{V}^+)$ for every α-vector in $\mathcal{V}^+$ and remove those α-vectors that are completely dominated by others [Sondik 1971][Monahan 1982].

Because each α-vector is associated with some action, the policy can be executed by taking the action relative to the best α-vector at the current belief state b. Equation 25 can give us a policy for all belief states, by taking the argmax over the α-vectors and finding out to which action the vector corresponds. Thus, a set Γ of α-vectors can represent a value function from which an agent can extract a policy π.

In a POMDP, calculating the optimal policy is not easy and the belief space, which helps the agent to reason over it's uncertainty, is actually the cause of this difficulty. The primary concern is the continuous nature of this belief space. Although a number of algorithms that successfully produce optimal policy solutions in this space have been developed over the years, these techniques are too complex and computationally inefficient [Papadimitriou and Tsitsiklis 1987]. This is due to the fact that the set of

α-vectors grows exponentially with every algorithm iteration. This exponential growth factor is a result of the computational cost of each iteration depending on the number of vectors that are in Γ. This causes the various algorithms to become expensive.

Another difficulty is the *curse of dimensionality* [Smith 2005]. This refers to the issue that the dimension of the belief space is equal to the number of states in the POMDP problem. This causes the size of this space to grow exponentially with the number of states. For example, if a POMDP problem is characterised with a 100 states, then the resulting belief space will have 100 dimensions. Because we maintain continuous distributions over this high dimensional space, solving POMDPs exactly is computationally intractable.

Lastly, the *curse of history* can also cause the computation of an optimal policy to be difficult [Png 2011]. This is a result of long planning horizons. The number of created beliefs increases exponentially with respect to the planning horizon [Pineau, Gordon, and Thrun 2006][Silver and Veness 2010][Lim, Hsu, and Lee 2011]. Therefore, the complexity of planning grows exponentially with the time horizon and can be particularly detrimental to computation times as the majority of planning tasks require an agent to take many actions before it eventually reaches its goal.

2.3.4 Value Iteration

Because a POMDP can be treated as a continuous space MDP, where the continuous space is the belief space (see Section 2.3.2), value iteration can therefore also be used to compute the value function in this paradigm. The value iteration algorithm for a POMDP problem has the same basic structure as the algorithm used for completely observable discrete MDP problems presented in Section 2.2.5. Value iteration computes $\mathcal{V}_{t+1}$ (the parsimonious representation of $V_{t+1}(b)$) from $\mathcal{V}_t$ (the parsimonious representation of $V_t(b)$).

The value function at time t can be computed by solving the equation

$$V_{t+1}(b) = \max_{a \in A} \left[R^b(b,a) + \gamma \sum_{b' \in \mathcal{B}} T^b(b,a,b') V_t(b') \right] \tag{27}$$

where

$$R^b(b,a) = \sum_{s \in S} b(s) R(s,a), \tag{28}$$

$$T^b(b,a,b') = \sum_{s' \in S} Z(s',a,o) \sum_{s \in \mathcal{S}} T(s,a,s') b(s) \tag{29}$$

and the value function at time $t = 0$ is initialised to $V_0(b) = 0, \forall b \in \mathcal{B}$. Rewriting Equation 27 in terms of the original POMDP formulation equates to

$$V_{t+1}(b) = \max_{a \in A} \left[\sum_{s \in S} b(s) R(s,a) + \sum_{o \in \mathcal{O}} P(o|b,a) V_t(\tau(b,a,o)) \right] \tag{30}$$

where $\tau(b, a, o) = b^{a,o}(s') = b'$ and $P(o|b, a)$ is given in Equation 20.

The most naive way to construct $\mathcal{V}_{t+1}^{+}$ is by enumerating all the possible actions and observation mappings from $\mathcal{V}_t^{+}$, which results in a set that is $|A||\mathcal{V}_t^{+}|^{|\mathcal{O}|}$ in size. However, many vectors in $\mathcal{V}_t^{+}$ may be dominated by others. So, to compute $\mathcal{V}_{t+1}^{+}$ we actually only need to consider $\mathcal{V}_t$. A full Bellman backup is used to compute $\mathcal{V}_{t+1}^{+}$ from $\mathcal{V}_t$ which involves finding the solution to

$$\mathcal{V}_{t+1}^{+} = \bigcup_{a \in A} \mathcal{V}_t^{a} \tag{31}$$

where

$$\mathcal{V}_t^{a} = \bigoplus_{o \in \mathcal{O}} \mathcal{V}_t^{a,o}, \tag{32}$$

$$\mathcal{V}_t^{a,o} = \left\{ \frac{1}{|\mathcal{O}|} r_a + \alpha^{a,o} : \alpha \in \mathcal{V}_t \right\}, \tag{33}$$

$$\alpha^{a,o}(s) = \sum_{s' \in S} Z(s', a, o) T(s, a, s') \alpha(s'), \tag{34}$$

$V_1 \oplus V_2 = \{\alpha_1 + \alpha_2 \mid V_1 \in \alpha_1, V_2 \in \alpha_2\}$, r_a is a vector representation of the reward function, $\mathcal{V}_t$ is the vector set prior to the backup and $\mathcal{V}_{t+1}^{+}$ is the the new vector set after the backup [Sondik 1978][Cassandra et al. 1997][Guy et al. 2013].

Because Equation 34 generates $|V| \times |A| \times |\mathcal{O}|$ vectors which requires $|S|^2$ operations and Equation 32 generates $|V|^{|\mathcal{O}|}$ vectors for each action, with each new vector requiring $|S|$ operations, the overall complexity of a single iteration is therefore

$$O\left(|V| \times |A| \times |\mathcal{O}| \times |S|^2 + |V|^{|\mathcal{O}|} \times |A| \times |S|\right).$$

This shows that the set of α-vectors grows exponentially with every iteration, resulting in a prohibitively expensive algorithm as the computational cost of each iteration depends on the number of vectors in V.

Some methods compute $\mathcal{V}_{t+1}$ by first constructing $\mathcal{V}_{t+1}^{+}$ and then performing pruning to remove the dominated α-vectors from $\mathcal{V}_{t+1}^{+}$ [Monahan 1982][Cassandra 1997]. Other algorithms compute $\mathcal{V}_{t+1}$ directly from $\mathcal{V}_t$, omitting the consideration of useless α-vectors [Sondik 1971][Smallwood and Sondik 1973][Cheng 1988][Kaelbling et. al. 1998]. Although these methods result in more manageable value functions [Cassandra et al. 1997][Littman 1996], they are still not sufficient for scaling to domains with more than a few dozen states. Because these methods are feasible only for problems with a small number of states, they are often disregarded.

Using existing exact POMDP algorithms to compute a continuous and high dimensional value function defined over the belief space for problems with a large number of states is considered to be intractable. However, approaches which compute approximate value functions have been formulated in an effort to develop algorithms which can solve many real-world problems [Murphy 2000].

2.4 Point-Based Partially Observable Markov Decision Process Algorithms

Over the past years, POMDP solving algorithms have seen significant advancement as modern solvers can now solve POMDP problems with high complexity and an extensive number of states. This breakthrough largely stemmed from computing value functions over a finite subset of the belief space. Algorithms which compute a value function in this way are referred to as point-based POMDP algorithms.

These algorithms use value iteration as other POMDP algorithms do, using the same style of value iteration, but the details are different, as will be discussed in subsequent sections. However, because they only consider a subset of the belief space, their value iteration techniques can be seen to exhibit slight changes when compared to the traditional approach (see Section 2.4.4).

The value iteration algorithm for point-based POMDP solvers is known as point-based value iteration (PBVI) and allows us to solve larger POMDP tasks significantly faster [Pineau et al. 2003]. PBVI algorithms explore the belief space, focusing on reachable belief states from some initial belief, and apply the point-based backup operator (see Equation 42) in order to maintain a value function. One point-based algorithm differs from another due mainly to their differences in the manner in which they select their core subset of beliefs and the order by which the value at those belief states are updated.

The PBVI algorithm allows us to approximately compute a solution to large POMDPs rapidly. We introduce this approach and give a discussion about its details in this section. Firstly, we present the basic insight that point-based POMDP algorithms take advantage of and follow to give a thorough description of the workings of the approach.

2.4.1 Limiting the Size of the Value Function

As mentioned earlier in Section 2.3.3 and 2.3.4, it is fundamentally important to limit the number of vectors that represent the value function when performing value iteration using POMDPs. However, there is a trade-off between the accuracy of the value function and the number of vectors that are used to represent the value function. By avoiding exponential growths with regards to computation times by decreasing the number of vectors that represent a value function, we may potentially compromise its accuracy. Therefore, establishing a means of efficiently choosing which vectors to remove from the complete set of vectors is vital.

We can alleviate this issue by collecting a set of reachable beliefs from an initial belief state $b_0 \in \mathcal{B}$, where $\mathcal{B}$ is the entire belief space, and maintaining the value function only over these beliefs [Hauskrecht 2000]|[Pineau et al. 2003]. The set of reachable beliefs $\mathcal{R}(b_0)$ can be obtained by applying the belief update procedure, given in Equation 14-19, starting from the initial belief state b_0. These points can be attained by performing the belief update procedure starting at b_0 and in turn deciding which sequence of actions and observations should be picked in order to gather a set of belief points that will

allow us to find an approximately optimal solution. The following section, Section 2.5, describes a variant of the point-based approach, kown as the SARSOP algorithm, as this is the point-based POMDP algorithm instantiation that we use in this work.

2.4.2 Value Function Updates

When we update the value function V at a finite set of belief points B, we do not need to use a full Bellman backup (see Equations 31-34). By manipulating the value function update procedure, we can use a less expensive solution to update the set of vectors representing the value function. We refer to the set of vectors representing V as $\mathcal{V}$. With Equations 35-41, we show how the value at a belief point $b \in B$ after a Bellman backup over a given value function V is computed and show how the calculation of this value can be used to calculate the new α-vector that would have been optimal for b, if we had ran the complete Bellman backup.

$$V'(b) = \max_{a \in A} R(b,a) + \gamma \sum_{o \in \mathcal{O}} P(o|b,a) V(b^{a,o}) \tag{35}$$

$$= \max_{a \in A} R(b,a) + \gamma \sum_{o \in \mathcal{O}} P(o|b,a) \max_{\alpha \in \mathcal{V}} \alpha \cdot b^{a,o} \tag{36}$$

$$= \max_{a \in A} R(b,a) + \gamma \sum_{o \in \mathcal{O}} P(o|b,a) \max_{\alpha \in \mathcal{V}} \sum_{s' \in S} \alpha(s') \cdot b^{a,o}(s') \tag{37}$$

$$= \max_{a \in A} R(b,a) + \gamma \sum_{o \in \mathcal{O}} P(o|b,a) \max_{\alpha \in \mathcal{V}} \sum_{s' \in S} \alpha(s') \frac{Z(s',a,o)}{P(o|b,a)} \sum_{s \in S} b(s) T(s,a,s') \tag{38}$$

$$= \max_{a \in A} R(b,a) + \gamma \sum_{o \in \mathcal{O}} \max_{\alpha \in \mathcal{V}} \sum_{s \in S} b(s) \sum_{s' \in S} \alpha(s') Z(s',a,o) T(s,a,s') \tag{39}$$

$$= \max_{a \in A} R(b,a) + \gamma \sum_{o \in \mathcal{O}} \max_{\alpha \in \mathcal{V}} \sum_{s \in S} b(s) \alpha^{a,o}(s) \tag{40}$$

$$= \max_{a \in A} R(b,a) + \gamma \sum_{o \in \mathcal{O}} \max_{\alpha \in \mathcal{V}} b \cdot \alpha^{a,o} \tag{41}$$

where $R(b,a) = \sum_{s \in S} b(s) R(s,a)$ and $\alpha^{a,o} = \sum_{s' \in S} \alpha(s') Z(s',a,o) T(s,a,s')$ [Guy et al. 2013]. Writing a more compact backup operation that generates a new α-vector for a particular belief point b is

$$backup(V,b) = \arg\max_{\alpha_a^b} b \cdot \alpha_a^b \tag{42}$$

where

$$\alpha_a^b = R(s,a) + \sum_{o \in \mathcal{O}} \arg\max_{\alpha^{a,o}} b \cdot \alpha^{a,o}. \tag{43}$$

The backup operation, given in Equation 42, is very similar to temporal difference methods in reinforcement learning [Mitchell 1997]. By expanding the belief state b up to a depth of one, the backup operation will backtrack the best α-vector at b for each

child. Then for each action a, it combines them using a Bellman equation so as to form one single hyperplane α_a^b. Thereafter, considering all the formed α_a^b-hyperplanes, the best hyperplane is added to $\mathcal{V}$.

Equation 42 implicitly prunes dominated vectors twice (as suggested by the two argmax expressions) which eliminates costly operations that generate an abundance of α-vectors. An important factor to note is that $\alpha^{a,o}$ is independent of b and can be stored and reused in the backup processes over other belief points.

Since the complexity of computing $\alpha^{a,o}$ (Equation 34) is $O\left(|S|^2\right)$, which is computed for every α-vector in $\mathcal{V}$, therefore the computation of all $\alpha^{a,o}$ has a complexity of

$$O\left(|S|^2 \times |A| \times |\mathcal{O}| \times |\mathcal{V}|\right).$$

Considering the summation and inner product operation of Equation 43 which has a complexity of $O\left(|S| \times |\mathcal{O}|\right)$ and the complexity of adding the reward vector which is $O\left(|S|\right)$, the complexity of computing α_a^b for every α-vector in $\mathcal{V}$ is therefore

$$O\left(|A| \times |S| \times |\mathcal{O}|\right).$$

Thus, the complexity of the point-based backup (Equation 42) is

$$O\left(|S|^2 \times |A| \times |\mathcal{O}| \times |\mathcal{V}| + |A| \times |S| \times |\mathcal{O}|\right).$$

Because the $\alpha^{a,o}$ are independent of the current belief b, a full point-based backup over the belief subset B, equating to $|B|$ multiplied by the complexity of a single point-based backup, will not be necessary. Thus, the complexity of a full backup for the subset B, where every $\alpha^{a,o}$ is computed only once and cached, is

$$O\left(|S|^2 \times |A| \times |\mathcal{O}| \times |\mathcal{V}| + |B| \times |A| \times |S| \times |\mathcal{O}|\right).$$

Comparing the $O\left(|S|^2 \times |A| \times |\mathcal{O}| \times |\mathcal{V}| + |B| \times |A| \times |S| \times |\mathcal{O}|\right)$ complexity of the full point-based backup (Equation 42) with the $O\left(|V| \times |A| \times |\mathcal{O}| \times |S|^2 + |V|^{|\mathcal{O}|} \times |A| \times |S|\right)$ complexity of a single iteration of the exact backup (Equation 31), we can observe the advantages of using point-based backups. They decrease the costs of backup operations and ultimately improve the performance of POMDP algorithms.

2.4.3 Policy Execution

A policy obtained from a value function defined over a finite subset of the belief space can be determined by solving

$$\pi^V(b) = \arg\max_a R(b,a) + \gamma \sum_{o \in \mathcal{O}} P(o|b,a) V(b^{a,o}). \tag{44}$$

In order to efficiently determine the action to take with regards to a policy, we use the α-vector representation and label the resulting vectors from the point-based backup (Equation 42) using the action that generated it. In other words, we label the vectors according to the action that was used to construct the α-vector. We can then use $\max_{\alpha \in V} \alpha \cdot b$ to find the best α-vector for the current belief point b and execute the action corresponding to that vector.

2.4.4 Value Iteration

Point-based algorithms generally use the ideas of bounding the size of the value function discussed in Section 2.4.1 and optimising the value function using the point-based procedure described in Section 2.4.2. In essence, they all follow the PBVI framework presented in Algorithm 1.

The algorithm is composed of two primary components. The first is the collection of the belief space subset B, and the second is the update of the value function V. Point-based approaches differ in the manner in which they achieve these two components, while the stopping criterion typically depends on the choice of these two components. A time-dependent stopping criterion only exists among algorithm variants which continually improves their value function with time, allowing them to have an anytime nature.

Algorithm 1 Generalised Point-Based Value Iteration.

1: **while** stopping criterion is not reached **do**
2: Collect finite belief subset B from the belief space (see Section 2.4.1)
3: Update V over B (see Equation 42)

2.4.5 Initialising the Value Function

As with every value iteration method, we require an initial value function on which to base value function updates. When choosing an initial value function for PBVI, it is important to begin with some initial function that is as close as possible to the optimal value function V^*. By achieving this, we decrease the number of iterations that will be performed before convergence is reached. Some point-based algorithms require the initial value function to be a lower bound on V^*. An example of a lower bound value function $\underline{V}$ can be seen to be

$$\underline{V} = \{\alpha_{min}\} \tag{45}$$

where

$$\alpha_{min}(s) = \frac{\min_{s,a} R(s,a)}{1-\gamma} \tag{46}$$

which is equivalent to collecting the minimum reward at each time step. Other methods do exist for calculating a lower bound value function that is closer to V^* [Hauskrecht 1997][Smith et al. 2005].

Some point-based approaches also require an upper bound on the value function and the same considerations can be used when initialising this function. However, when initialising the upper bound, the belief-value representation is used over the vector representation. A belief-value representation maintains a value for every belief that is encountered. It is a mapping of belief states to values and are points on the convex envelope of the current estimate of the value function. For the belief states whose

mapping is not currently maintained over the over the convex value function, we are required to interpolate the value of those beliefs.

This representation is preferred mainly because when implementing value iteration using the vector represented upper bound, new vectors will typically have a lower value than currently existing vectors and so adding these vectors will have no effect on the existing value function. Different initialisation strategies have been developed over past years [Hauskrecht 1997][Hauskrecht 2000]. The simplest strategy to initialise the upper bound on the value function $\overline{V}$ is by assuming full observability and solving the MDP version of the problem. Using the underlying MDP optimal value function as an upper bound initialisation is a well known idea [Littman 1996]. Its suggests using the optimal Q-values (see Sections 2.2.4) of the underlying MDP to form the Q_{MDP} value function for POMDPs which is defined to be

$$Q_{MDP}(b) = \max_{a} Q(s,a)b(s). \tag{47}$$

If we want to use a tighter upper bound on the value function, we can consider the fast informed bound initialisation technique [Hauskrecht, 2000]. The fast informed bound initialiser updates the value function for each state s using

$$\overline{V}(b) = \max_{a \in A} \left[R(b,a) + \gamma \sum_{o \in \mathcal{O}} \max_{\alpha} P(o|b,a)\alpha(s') \right]. \tag{48}$$

2.4.6 Core Parameters

There are a number of parameters that influence the resulting value function of point-based algorithms and ultimately affect their performance. Typically, these parameters result in trade-offs between approximation accuracy and computational effort.

One parameter of concern is the number of belief points in B (the subset of the belief space $\mathcal{B}$) because the value update time largely depends on the size of B, while the accuracy of the value function also depends on the number of sampled belief points. The sampled belief points are the belief states that are chosen from the belief space and used to approximate the value function (see Section 2.4.1). Thus, we can observe a trade-off between satisfactory approximation and time taken to perform value function updates. As a result, point-based approaches control the number of belief points that comprise B.

Another parameter is the number of times the point-based backup operation (Equation 42) is performed in each value function update. In most cases, the value function can be updated without using all the belief points in B, but in some cases numerous backups over a single belief in an iteration can be profitable. Decreasing the number of backups reduces the computational effort, while increasing the number of backups yields a better approximation of V^*.

Pruning dominated vectors, in an attempt to maintain a compact representation of a value function, is also a vital aspect of point-based algorithms. Its importance is

evident in that the complexity of the point-based backup depends on the number of α-vectors that are used to represent the value function. Because of the inability of exact POMDP solvers to maintain a small number of vectors to represent the value function, it is possibly the main reason for their failures when they are required to solve problems in larger domains.

2.5 Successive Approximations of the Reachable Space under Optimal Policies (SARSOP)

As explained in Section 2.3.4, exact POMDP solvers are computationally intractable when they are implemented for solving tasks in real environments. This is because some tasks can contain up to or even more than 10^5 states which can cause their computational complexity to be expensive. The SARSOP algorithm is an approximate POMDP algorithm that improves computational efficiency and is practical for many applications in robotics [Kurniawati et. al 2008]. SARSOP exploits the idea of *optimally reachable belief spaces* to bring about these computational improvements and has been successfully applied to various robotics tasks such as coastal navigation, mobile robot exploration and grasping [Kurniawati et al. 2008].

Recent point-based POMDP approaches sample belief points only from the subset of reachable belief points from an initial belief $b_0 \in \mathcal{B}$ under an arbitrary sequence of actions, referred to as $\mathcal{R}(b_0)$ [Pineau et al. 2003][Smith and Simmons 2004][Spaan and Vlassis 2004][Smith and Simmons 2005[Hsu et al. 2008]. SARSOP pushes further in this direction by sampling beliefs near the subset of belief points reachable from b_0 under an optimal sequence of actions, known as $\mathcal{R}^*(b_0)$. However the optimal sequences of actions are not known in advance as this makes up the POMDP solution. Generally, it is believed that $\mathcal{R}(b_0)$ is much smaller than the size of $\mathcal{B}$, and $\mathcal{R}^*(b_0)$ is usually much smaller than $\mathcal{R}(b_0)$ i.e. $\mathcal{R}^*(b_0) \subset \mathcal{R}(b_0) \subset \mathcal{B}$ as seen in Figure 10.

Instead of directly approximating V^* or searching for π^* as other common approaches do, the SARSOP algorithm (Algorithm 2) focuses on finding an approximate cover of the space reachable under an optimal policy $\mathcal{R}_{\pi^*}(b_0)$ through sampling. Because multiple optimal policies may exist, SARSOP aims to sample $\mathcal{R}^*(b_0) = \bigcup_{\pi^*} \mathcal{R}_{\pi^*}(b_0)$ which is the union of all optimally reachable spaces. Therefore, having knowledge of $\mathcal{R}^*(b_0)$ is in some sense equivalent to knowing the POMDP solution (see Section 2.5.2). Thus the goal, here, is to approximate $\mathcal{R}^*(b_0)$.

SARSOP successively computes approximations of $\mathcal{R}^*(b_0)$ and converges to it iteratively. To achieve this, it relies on heuristic exploration to sample $\mathcal{R}(b_0)$ and improves samples over time using a basic online learning method (see Section 2.5.2). It then avoids sampling in regions that are unlikely to be optimal by implementing a bounding technique. By focusing sampling on the region near $\mathcal{R}^*(b_0)$, SARSOP is able to greatly reduce the computational effort and ultimately results in a more efficient approach.

The value function is represented using a set Γ of α-vectors, where each α-vector in Γ must dominate all other vectors in Γ at some sampled belief. Pruning sampled beliefs that are not part of $\mathcal{R}^*(b_0)$ reduces the size of Γ and in turn further improves

computational efficiency. We discuss this aspect in more detail in Section 2.5.3.

In order to simplify the notation of $\mathcal{R}(b_0)$ and $\mathcal{R}^*(b_0)$, we note that $\mathcal{R}$ and $\mathcal{R}^*$ refer to $\mathcal{R}(b_0)$ and $\mathcal{R}^*(b_0)$ respectively.

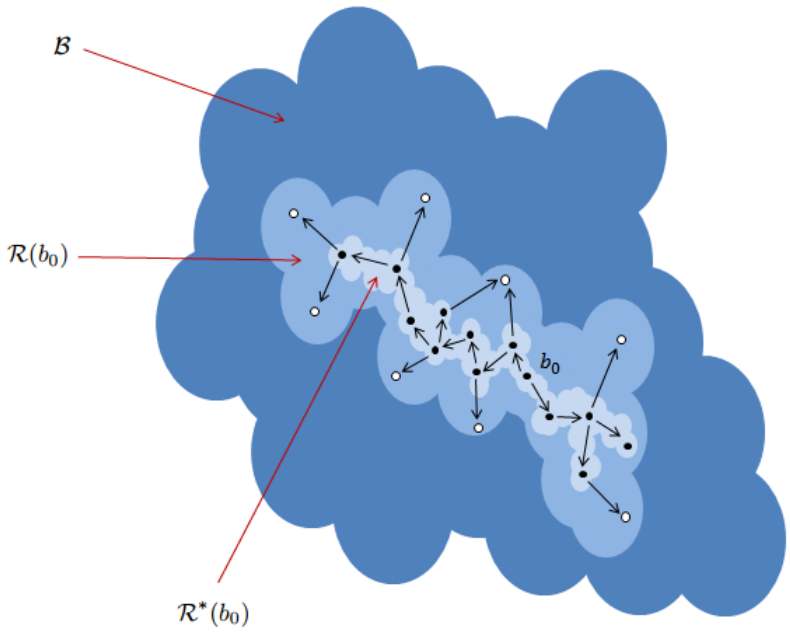


Figure 10: A Graphical representation of the Belief Space $\mathcal{B}$, Reachable Belief Space $\mathcal{R}(b_0)$ and Optimally Reachable Belief Space $\mathcal{R}^*(b_0)$.

2.5.1 Algorithm Overview

The SARSOP algorithm (Algorithm 2) uses three main functions over which it iterates. These functions are SAMPLE (Algorithm 4), BACKUP (Algorithm 3) and PRUNE (see Section 2.5.3).

As with all point-based approaches, SARSOP collects a finite subset of belief points from the belief space $\mathcal{B}$. The sampled belief points form a tree $T_{\mathcal{R}}$, with each node in $T_{\mathcal{R}}$ representing a point that has been sampled (see Figure 11). The initial belief point b_0 is the root of $T_{\mathcal{R}}$. The same symbol b is used to refer to both a sampled point and its corresponding node in $T_{\mathcal{R}}$ so as to avoid confusion. In order to sample a new belief point, we use suitable probability distributions or heuristics to choose a node b from the tree $T_{\mathcal{R}}$ from which to sample, along with an action $a \in A$ and observation $o \in \mathcal{O}$ (see Section 2.5.2). We then compute the new sampled belief point b' using Equation 14-19 and insert it into $T_{\mathcal{R}}$ as a child of b.

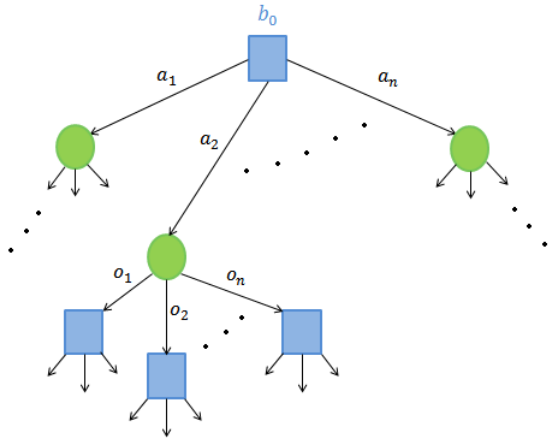


Figure 11: The Formation of Belief Tree $T_{\mathcal{R}}$ Rooted at b_0.

The algorithm maintains both a lower bound $\underline{V}$ and an upper bound $\overline{V}$ on the optimal value function V^* to focus sampling near $\mathcal{R}^*$. In this way SARSOP is able to bias sampling towards $\mathcal{R}^*$(see Section 2.5.2). The lower bound $\underline{V}$ is represented using a set Γ of α-vectors and can be initialised in various ways, for example using a fixed-action policy or a blind policy [Hauskrecht 2000]. On the other hand, the MDP, the Fast Informed Bound technique or the sawtooth approximation can be used to initialise $\overline{V}$ [Hauskrecht 2000].

Suppose we consider an agent acting in a world with only two states as we did in Section 2.3.3. As discussed in Section 2.3.3, the value functions can therefore be represented using linear segments. Figure 12 gives a graphical representation of a lower bound $\underline{V}$ and an upper bound $\overline{V}$ on an optimal value function $V^*(b)$ over the belief space. The arrows in Figure 12 point to the upper bound value $\overline{V}(b)$ and lower bound value $\underline{V}(b)$ at belief b, while the black circle refers to the actual optimal value at b.

Algorithm 2 SARSOP.

1: Initialise the lower bound $\underline{V}$ on the optimal value function V^* and the upper bound $\overline{V}$ on V^*.
2: Insert the initial belief point b_0 as the root of the tree $T_{\mathcal{R}}$.
3: **while** termination condition is not reached **do**
4: SAMPLE($T_{\mathcal{R}}$, Γ) (see Algorithm 4).
5: Select a subset of nodes from $T_{\mathcal{R}}$ and perform BACKUP($T_{\mathcal{R}}$, Γ, b) on each selected node b (see Algorithm 3).
6: PRUNE($T_{\mathcal{R}}$, Γ) (see Section 2.5.3).
7: **return** Γ.

Backup operations are performed on selected nodes in $T_{\mathcal{R}}$. A backup operation at a node b collects and combines the information in the children of a node b and propagates

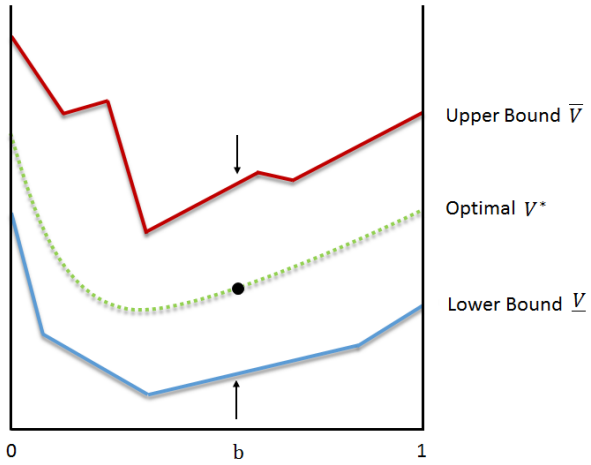


Figure 12: An example of a lower bound and upper bound on an optimal value function $V^*(b)$.

it back to b. The standard α-vector backup (Algorithm 3) is performed on the lower bound of the value function, while the Bellman backup (Equation 42) is performed on the upper bound. The α-vector backup produces the same value function approximation at b as the Bellman backup, but while the α-vector backup propagates the gradient of the value function approximation as well as the value, the Bellman backup propagates only the value. The α-vector backup propagates this gradient and value so that it can obtain a global approximation over the entire belief rather than a local approximation at b.

Algorithm 3 α-Vector Backup at a Node b.

BACKUP($T_{\mathcal{R}}$, Γ, b)

1: $\forall a \in A$, $\forall o \in \mathcal{O}$, $\alpha_{a,o} \leftarrow \arg\max_{\alpha \in \Gamma} \alpha \cdot b'$

2: $\forall a \in A$, $\forall s \in S$, $\alpha_a(s) \leftarrow R(s,a) + \gamma \sum_{o \in \mathcal{O}} \sum_{s' \in S} Z(s', a, o) T(s, a, s') \alpha_{a,o}(s')$

3: $\alpha' \leftarrow \arg\max_{\alpha} \alpha_a \cdot b$

3: Insert α' into Γ

When SAMPLE and BACKUP are invoked, new sampled points and α-vectors are generated. However, not all of the α-vectors may be useful for constructing an optimal policy and so those α-vectors that are dominated by other α-vectors are pruned to improve computational efficiency.

SARSOP is an anytime approach that returns the best policy given a pre-specified amount of time. Because the algorithm gradually reduces the gap Φ between the lower and upper bound on the value function at b_0, it can either use a target gap size or time limit as its termination condition.

2.5.2 Sampling

When sampling new belief points (see Algorithm 4), a target gap size Φ between the lower and upper bound is set at the root b_0 of tree $T_{\mathcal{R}}$. SARSOP traverses down a single path of $T_{\mathcal{R}}$ by selecting at each node the action a with the highest upper bound and observation o that makes the biggest contribution to decreasing the gap at the root of $T_{\mathcal{R}}$. However, the sampling path may be terminated under suitable conditions. Thus, the techniques for selecting actions and observations, together with the choice of termination conditions control the resulting sampling distribution.

Algorithm 4 Sampling.

SAMPLE($T_{\mathcal{R}}$, Γ)

1: Set L to the current lower bound on the value function at the root b_0 of tree $T_{\mathcal{R}}$ and set U to $L + \Phi$, where Φ is the target gap size at the root.
2: SAMPLEPOINTS($T_{\mathcal{R}}$, Γ, b_0, L, U, Φ, 1)

SAMPLEPOINTS($T_{\mathcal{R}}$, Γ, b, L, U, Φ, h)

3: **if** $\hat{V} \leq L$ and $\overline{V}(b) \leq \max\left\{U, \underline{V}(b) + \Phi\gamma^{-h}\right\}$, where $\hat{V}$ is the prediction of $V^*(b)$ **then**
4: **return**
5: **else**
6: $\underline{Q} \leftarrow \max_a \underline{Q}(b, a)$
7: $L' \leftarrow \max\left\{L, \underline{Q}\right\}$
8: $U' \leftarrow \max\left\{U, \underline{Q} + \Phi\gamma^{-h}\right\}$
9: $a' \leftarrow \arg\max_a \overline{Q}(b, a)$
10: $o' \leftarrow \arg\max_o P(o|b, a')\left[\overline{V}(\tau(b, a', o)) - \underline{V}(\tau(b, a', o))\right]$
11: $L_h \leftarrow \dfrac{L' - \sum_{s\in S} R(s, a')b(s) - \gamma \sum_{o\neq o'} P(o|b, a')\underline{V}(\tau(b, a', o))}{\gamma P(o'|b, a')}$
12: $U_h \leftarrow \dfrac{U' - \sum_{s\in S} R(s, a')b(s) - \gamma \sum_{o\neq o'} P(o|b, a')\overline{V}(\tau(b, a', o))}{\gamma P(o'|b, a')}$
13: $b' \leftarrow \tau(b, a', o')$
14: Insert b' into $T_{\mathcal{R}}$ as a child of b
15: SAMPLEPOINTS($T_{\mathcal{R}}$, Γ, b', L_h, U_h, Φ, $h + 1$)

One criterion to stop sampling is to terminate a sampling path when it reaches a node whose gap between the lower and upper bound is smaller than $\gamma^{-h}\Phi$, where h is the depth of the node in $T_{\mathcal{R}}$ (the height of the node in $T_{\mathcal{R}}$) [Smith and Simmons 2005]. This seems advantageous because if each leaf of $T_{\mathcal{R}}$ has a gap smaller than $\gamma^{-h}\Phi$, then we are guaranteed that the gap at the root will be smaller than Φ. However, as the target gap Φ at the root decreases, the sampling path must traverse deeper down the

tree. But, the size of the set of points in $\mathcal{R}$ increase much faster than those in $\mathcal{R}^*$ as the sampling path traverses deeper down the tree. As a result, sampling beliefs near $\mathcal{R}^*$ becomes increasingly difficult as they are sparse in $\mathcal{R}$. Therefore, we hope to achieve a shallow sampling path while still being able to reach the target gap Φ at the root of $T_{\mathcal{R}}$. Keeping the sampling path as shallow as possible brings about a potential dilemma in that some nodes with high expected rewards lie deep in the tree. Thus, although we ideally hope to keep a shallow sampling path, we must allow the sampling path to traverse deep enough in order to reach them.

Selective Deep Sampling

SARSOP implements a technique known as *selective deep sampling* to ensure that nodes with high expected rewards that may lie deep in the tree are reached [Kurniawati et al. 2008]. Because each backup operation selects the action that maximises the expected reward, lower bound improvements are rapidly propagated to the root of the tree when nodes with high expected rewards are reached. This quickly provides the necessary information to stop sampling in regions that are likely to be outside of $\mathcal{R}^*$ and also directly improves the policy. SARSOP gives preference to lower bound improvements and continues down a sampling path beyond a node with a gap of $\gamma^{-h}\Phi$ only if a prediction is made that doing so will lead to lower bound improvements at the root.

Conceptually a prediction is achieved by predicting an optimal value $V^*(b)$ at a node b and propagating the predicted value $\hat{V}$ upwards towards the root of $T_{\mathcal{R}}$. We expand b if $\hat{V}$ improves the lower bound at the root and repeat the process at the next chosen node down the sampling path. Otherwise, if the prediction shows no improvement to the lower bound at the root, we proceed to check the target gap size Φ at the root to decide whether or not to terminate the sampling.

In order to compute $\hat{V}$, the predicted value of the optimal value $V^*(b)$, SARSOP uses a basic learning method which clusters collected belief points according to suitable features and uses previously computed values of beliefs in the same cluster as b to predict the value of b. The features used to cluster the beliefs are the initial upper bound and the entropy of b. These features discretise the belief space into a finite number of groups and allow us to learn which parts of the belief space are worth exploring. Any new belief belonging to a particular group will have a predicted value equal to the average value of the beliefs in that group. On the contrary, if a new belief forms a new group, the initial upper bound of the new belief is used as the predicted value.

Figure 13 gives a graphical representation of how these groups of beliefs are formed based on their initial upperbound value and entropy. Say, for example, belief D in Figure 13 is a new belief entered into group 9. Therefore, the predicted value $\hat{V}$ of belief D will be its initial upper bound value. However, if belief C is a new belief entered into group 17, the predicted value $\hat{V}$ of belief C will then be equal to the average value between belief A and belief B.

For increased efficiency, SARSOP does not propagate the predicted value to the root in a literal sense, but instead passes a lower bound target level L down the sampling

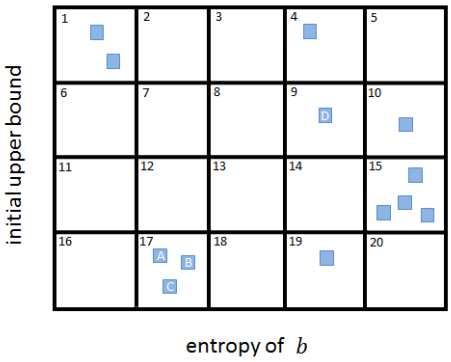


Figure 13: Clustering of beliefs based on their initial upper bound and entropy.

path where the predicted value $\hat{V}$ is checked against L. If $\hat{V}$ fails to meet L at a node b, the lower bound at node b will not be propagated further up towards the root of the tree.

There is a need to compute an intermediate target level L' for action a because the value function information is propagated from b' to b only when the action that takes b to b' has a greater value than all other actions at b. An intermediate target level L' for action a is calculated and set to be the maximum over L and the values of all actions at node b (see line 6 and 7 of Algorithm 4). The lower bound on the value of action a is computed using

$$\underline{Q}(b,a) = \sum_{s \in S} R(s,a)b(s) + \gamma \sum_{o \in \mathcal{O}} P(o|b,a)\underline{V}(b') \tag{49}$$

while the target level for b' is the value required for $\underline{Q}(b,a)$ to achieve its target L' (see line 11 of Algorithm 4). In order for the algorithm to protect itself against misleading predictions that are caused by unnecessarily deep sampling paths, SARSOP continues down a sampling path until the gap between the upper and lower bounds is $k\gamma^{-h}\Phi$ for some $k < 1$.

Gap Termination Criterion

As we've mentioned, it is insufficient for SARSOP to require a gap size $\gamma^{-h}\Phi$ for all leaves of $T_{\mathcal{R}}$. However, we can observe that it is sufficient to ensure that the condition is satisfied somewhere along all the paths from the root to the leaves, instead of at the leaves themselves. This allows the algorithm to terminate a sampling path as early as possible. This not only leverages information globally, but also improves computational efficiency.

To ensure that the condition is satisfied somewhere along the paths from root to leaves, SARSOP also passes an upper bound target level U down the sampling path. For a node b at depth h, a sampling path may be terminated if its upper bound is lower than $\underline{V}(b) + \gamma^{-h}\Phi$ or the upper bound target level U passed down from the parent of b

(see line 3 of Algorithm 4). This has the same effect as requiring all leaves to have a gap of no more than $\gamma^{-h}\Phi$ and will therefore ensure that the target gap size of Φ at the root b_0 is achieved. The upper bound target level U is passed down a sampling path in a similar fashion to that of the lower bound target level L as seen in lines 8 and 12 of Algorithm 4.

The combination of deep sampling and gap termination criterion produce an effective sampling scheme that travels deep into $T_{\mathcal{R}}$ if needed, giving a better approximation towards $\mathcal{R}^*$ and avoiding unnecessary samples in $\mathcal{R} \backslash \mathcal{R}^*$.

2.5.3 Pruning

Some existing point-based algorithms prune α-vectors from the set Γ of α-vectors if a vector is dominated by others over the entire belief space $\mathcal{B}$ [Pineau et al. 2003][Smith and Simmons 2004][Spaan and Vlassis 2004][Smith and Simmons 2005[Hsu et al. 2008]. However, the idea of an optimally reachable belief space suggests an alternative and more aggressive pruning method. This is achieved by pruning α-vectors that are dominated by others over $\mathcal{R}^*$, instead of $\mathcal{B}$.

Because $\mathcal{R}^*$ is not known in advance, SARSOP requires a basis on which to compute its value function approximation. Therefore, by considering the set B of all sampled belief points, it is able to overcome this difficulty by using the tree $T_{\mathcal{R}}$ as an approximation of $\mathcal{R}^*$. It improves this approximation and ensures that the size of B is kept small by pruning away the points in B that are provably suboptimal and do not lie in $\mathcal{R}^*$. If $\overline{Q}(b, a_1) < \underline{Q}(b, a_2)$ for a node b in $T_{\mathcal{R}}$ and two actions a_1 and a_2, all the sampled points collected from the subtree formed by taking action a_1 at b are pruned because an optimal policy would never take that action at b and traverse the subtree beneath it. Although there may be other paths in $T_{\mathcal{R}}$ that could possibly lead to those pruned belief points under some optimal policy, the advantages of keeping the size of B small usually outweigh the loss in approximation quality brought about by over-pruning. However, these belief points can be recovered with time from other paths in $T_{\mathcal{R}}$.

The pruning of belief points in turn enables a more aggressive α-vector pruning strategy. The SARSOP algorithm prunes an α-vector if it is dominated by others over B using an idea called δ-*dominance* [Kurniawati et al. 2008]. δ-Dominance imposes a requirement for dominance over a δ-neighbourhood (see Figure 14). Thus, when considering two α-vectors, α_1 and α_2, α_1 dominates α_2 at a belief point b if $\alpha_1 \cdot b' \geq \alpha_2 \cdot b'$ at every belief point b' whose distance to b is less than δ, where δ is some fixed constant. δ-Dominance can be checked very quickly by calculating the distance d from b to the intersection of the hyperplanes represented by α_1 and α_2 and ensuring that $d \geq \delta$. Figure 14 illustrates δ-dominance at a belief b. As a result of δ-dominance, we can observe that α_3 dominates α_2, but not α_1 in the δ-neighbourhood of b.

The simple dominance condition, which states that for two α-vectors, α_1 and α_2, α_1 dominates α_2 at a belief point b if $\alpha_1 \cdot b \geq \alpha_2 \cdot b$, is not used because the set B is a finitely sampled approximation of $\mathcal{R}^*$ and so a computed policy might choose an action that causes the algorithm to slightly veer away from $\mathcal{R}^*$. This could potentially

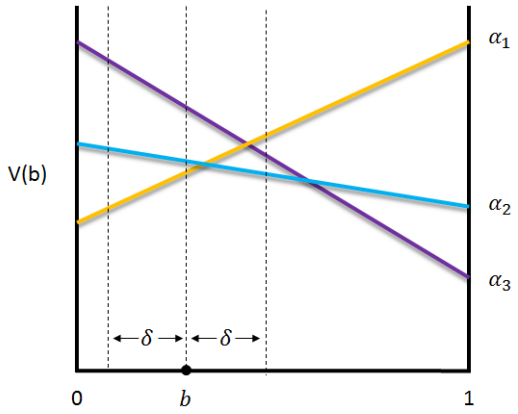


Figure 14: δ-Dominance.

occur as a result of SARSOP computing an approximately optimal policy over B only. If the algorithm veers away from $\mathcal{R}^*$, then it may end up in a region which causes it to generate poor approximations of the value function.

In general, computing the optimal policy for a POMDP model is computationally intractable. Approaches to dealing with this include computing an approximately optimal value function and extracting the policy from this approximation [Pineau et al. 2003][Smith and Simmons 2004][Kurniawati et al. 2008], as SARSOP does, or to develop algorithms that can exploit problem characteristics [Littman et al. 1995][Parr and Russell 1995][Zhang and Liu 1997]. Although POMDP planning algorithms have become more scalable and efficient over the years, using such methods in real-world domains can still be challenging [Smith 2005][Guy et al. 2013]. As a result, we leverage the transfer of knowledge from previous tasks to improve the learning rates and expensiveness of solving POMDP tasks. In order to achieve this, we consider *action priors,* which we discuss further in the following section (Section 2.6).

2.6 Action Priors

Choosing the best action to take in situations where agents have an array of actions can be a difficult task. However, the selection of actions may be guided by action priors as they provide information about the usefulness of actions under particular circumstances. Their purpose is to provide knowledge and give an intuition over the actions that make sense in particular scenarios. This knowledge is established by considering the statistics of action choices over the lifetime of an agent [Rosman and Ramamoorthy 2012][Rosman and Ramamoorthy 2015]. By considering MDPs and the reinforcement learning paradigm, we observe two methods for learning action prior knowledge. These are namely, state-based action priors and perception-based action priors.

Consider a domain defined by $D = (S, A, T, \gamma)$ where the state set S, action set A and transition function T is fixed for the entire domain. We define a task to be

$M = (D, R)$, a set of tasks to be $\mathcal{M} = \{M\}$ and the set of optimal policies corresponding to those tasks as $\Pi = \{\pi^*_M\}$. Note that the tasks in $\mathcal{M}$ vary only with the reward function. The goal is to discover a distribution over the action set for each state. This distribution is Dirichlet and is the probability of each action being used in a particular state by an optimal policy for any task in that domain. We say that an action $a_1 \in A$ is more useful than action $a_2 \in A$ in a state s, if a_1 is used by more optimal policies in s [Rosman and Ramamoorthy 2012].

2.6.1 State-based Action Priors

For each state, a count $\alpha_s(a)\ \forall a \in A$, representing the number of times an action is used by an optimal policy in that state, is maintained for each action. If $\alpha_s(a_1) = \alpha_s(a_2)$ for some state, then a_1 and a_2 are equally favourable in that state. The initial value associated with this count is denoted as $\alpha^0_s(a)$ and can be initialised to any value so as to portray prior knowledge. The α_s counts are updated using

$$\alpha^{new}_s(a) \leftarrow \begin{cases} \alpha_s(a) + 1 & \text{if } a = \arg\max_a Q^{new}(s, a) \\ \alpha_s(a) & \text{otherwise} \end{cases} \tag{50}$$

where $Q^{new}(s, a)$ is a new Q-function obtained from solving a new task [Rosman and Ramamoorthy 2012]. In this way, $\alpha_s(a)$ tells us how many times action a was considered the best action to take (i.e. the action that lead to the highest return) in state s in any Q-function.

The state-based action priors $\theta_s(a)$ are a state dependent probability distribution over the action set of an agent and are acquired by sampling from the Dirichlet distribution: $\theta_s(a) \sim \text{Dir}(\alpha_s)$. A Dirichlet distribution is a multivariate generalization of the beta distribution and is a conjugate prior of the categorical distribution and multinomial distribution [Galleguillos and Belongie 2010]. Since $\theta_s(a)$ is sampled as a probability distribution over the action set A, this means $\sum_a \theta_s(a) = 1,\ \forall s \in S$. This procedure of obtaining $\theta_s(a)$ from $\text{Dir}(\alpha_s)$ can be seen as a form of averaging Q-functions. However, it is not to be mistaken with naive averaging as this can cause problems often resulting in detrimental results.

Take for example an agent having to navigate to its goal, but is faced with a wall obstacle in front of its current state s. One Q-function could suggest that the agent should take the action that results in the agent moving around the wall to the right, while another Q-function could suggest that the agent should take action that results in the agent moving around the wall to the left. However, taking the average of these two Q-functions suggests that the agent should move forward into the wall. Thus, instead of using naive averaging, we infer that moving around the wall obstacle to the right and to the left are both feasible actions to choose, whereas moving forward is never considered to be the correct action to take. As a result, the action priors should put more weight on the actions that allow the agent to move around the wall than moving into it, reflecting the preferences elicited from the two Q-functions. The action priors learn this from experience by solving multiple tasks.

Action priors are particularly useful for providing an agent with knowledge (learned from experience) about which actions make sense in circumstances where the agent has an array of actions to choose from. Therefore, their use can be beneficial in seeding search in a policy learning process. Algorithm 5, known as ε-greedy Q-learning with State-based Action Priors (ε-QSAP), presents an algorithm that illustrates how these state-based action priors can be used with Q-learning [Rosman and Ramamoorthy 2012]. We note that when we have a uniform action prior, ε-QSAP is equivalent to the Q-learning baseline. In Algorithm 5, $\alpha^Q \in [0, 1]$ denotes the learning rate and $\varepsilon \in [0, 1]$ denotes the parameter that controls the trade off between exploration and exploitation. Note that the learning rate α^Q should not be confused with the α_s counts $\alpha_s(a)$. As Algorithm 5 iterates, both α^Q and ε are annealed after each episode.

The action selection step (Algorithm 5 line 5) consists of two cases and differentiates ε-QSAP from traditional Q-learning. The first case deals with exploiting the current policy stored in $Q(s, a)$ with probability $1 - \varepsilon$ and the second case deals with exploring other actions $a \in A$ with probability ε. In Q-learning, actions are typically chosen uniformly from the set of actions A during exploration. However, ε-QSAP chooses actions with probability based on the action prior $\theta_s(a)$ so as to base action selections during exploration on what were sensible action choices in previously solved tasks. Constructing the algorithm in this manner allows the agent to exploit its current estimate of the optimal policy with high probability and explores each action proportional to the number of times that action was considered to be a sensible choice in previous tasks.

Algorithm 5 ε-greedy Q-learning with State-based Action Priors (ε-QSAP).

Require: action prior $\theta_s(a)$

1: Initialise $Q(s, a)$ arbitrarily
2: **for** every episode $k = 1 \dots K$ **do**
3: Choose initial state s
4: **repeat**
5: $a \leftarrow \begin{cases} \arg\max_a Q(s, a) & \text{w.p. } 1 - \varepsilon \\ a \in A & \text{w.p. } \varepsilon\theta_s(a) \end{cases}$
6: Take action a, observe r, s'
7: $Q(s, a) \leftarrow Q(s, a) + \alpha^Q \left[r(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
8: $s \leftarrow s'$
9: **until** s is terminal
10: **end for**
11: **return** $Q(s, a)$

2.6.2 Perception-based Action Priors

Because there is a loss of generality with the use of state-based action priors in the sense that the re-use of this prior knowledge is state specific, an alternative approach is therefore required to generalise their use. This gives rise to the idea of perception-based

action priors. Primarily, there are two reasons to use perception-based action priors over those that are state-based. Firstly, by using perception-based action priors the domain need not be fixed and secondly, more data can be collected in cases where some states appear to be similar. So, in an effort to generalise action prior usage, we associate them with the perceptual information at a state rather than the state itself. Defining an observation space to be $\mathcal{O}$, we model perception-based action priors as $\theta_o(a), \forall o \in \mathcal{O}$ where observations are a function of a state, i.e. $o = o(s)$, and depend on the agent's sensory features.

Similarly to the state-based case, we have α_o counts. However they differ only in that they are observation dependent rather than depending on a particular state. These α_o counts are updated using

$$\alpha^{new}_{o(s)}(a) \leftarrow \begin{cases} \alpha_{o(s)}(a) + 1 & \text{if } a = \arg\max_a Q^{new}(s,a) \\ \alpha_{o(s)}(a) & \text{otherwise} \end{cases} \tag{51}$$

where $Q^{new}(s,a)$ is a new Q-function obtained from solving a new task [Rosman and Ramamoorthy 2012]. In this way, the rate at which learning occurs will increase because the action priors from multiple states will map to the same perception-based action priors. This is the result when the same observation is observed at different states and the increase in learning speeds is largely due to the fact that action prior information is updated better as more data is obtained for a particular observation during a training episode.

The perception-based action priors $\theta_o(a)$ are a context dependent probability distribution over the action set of an agent and are acquired by sampling from the Dirichlet distribution: $\theta_o(a) \sim \text{Dir}(\alpha_{o(s)})$. Since $\theta_o(a)$ is sampled as a probability distribution over the action set A, this means $\sum_a \theta_o(a) = 1, \forall o \in \mathcal{O}$.

Similarly to state-based action priors, the perception-based priors can be used with an adaptation of traditional Q-learning [Sutton and Barto 1998]. Algorithm 6, known as ε-greedy Q-learning with Perception-based Action Priors (ε-QPAP), presents an algorithm that illustrates how these perception-based action priors can be used with Q-learning [Rosman and Ramamoorthy 2012]. The difference between ε-QSAP and ε-QPAP is that with ε-QPAP, the perceptual action priors $\theta_o(a)$ are used instead of the state action priors $\theta_s(a)$. Line 5 of Algorithm 6 refers to acquiring perceptual information from the current state s. Note that the state information is global and unique, but the observations can seen to repeat throughout the domain. The perceptual action priors allow us to use different state spaces S and transition functions T to those used in the training process, as long as the observation space $\mathcal{O}$ remains consistent.

Although the actions are defined over the observation space, learning still occurs over the state space. In fact, because the action priors are defined over the observation space, they do not require the domain to be fully observable. Therefore, because only the perceptual information available to the agent is used, perception-based action priors can be applied to partially observable domains. As a result we can use action priors with the SARSOP algorithm presented in Section 2.5. In Section 3.3, we present three

variants of the SARSOP algorithm that use action priors which we believe will accelerate the learning of tasks in a POMDP context.

Algorithm 6 ε-greedy Q-learning with Perception-based Action Priors (ε-QPAP).

Require: action prior $\theta_o(a)$

1: Initialise $Q(s, a)$ arbitrarily
2: **for** every episode $k = 1 \dots K$ **do**
3: Choose initial state s
4: **repeat**
5: $o \leftarrow \text{observations}(s)$
6: $a \leftarrow \begin{cases} \arg\max_a Q(s,a) & \text{w.p. } 1-\varepsilon \\ a \in A & \text{w.p. } \varepsilon\theta_o(a) \end{cases}$
7: Take action a, observe r, s'
8: $Q(s,a) \leftarrow Q(s,a) + \alpha^Q \left[r(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$
9: $s \leftarrow s'$
10: **until** s is terminal
11: **end for**
12: **return** $Q(s, a)$

2.7 Conclusion

This chapter provides background on all the relevant material required to not only understand the research topic at hand, but also to give an idea of how we plan to tackle the research problem. It begins with a discussion of the details of Markov Decision Processes (MDPs). We introduce the framework of MDPs and the reinforcement learning paradigm as they are the building blocks of the concepts that are presented in this thesis. Reinforcement learning is the basis of what it means to compute an optimal policy in an MDP or POMDP and is presented to give an understanding of how action priors may be learned in a MDP setting.

Action priors are learned from experience and allow an agent to determine the actions that are sensible in situations. Although we illustrate an alternative approach for learning action priors in the POMDP setting later (see Section 3.3.1), it is essential for us to provide a description of how these are formulated using reinforcement learning as the priors learned from this approach will also be used in our experiments.

Following MDPs, we present a description of the Partially Observable Markov Decision Process (POMDP) framework. POMDPs are a powerful framework for planning under uncertainty, however their main disadvantage is their high computational complexity. As a result, many researchers have disregarded exact POMDP solvers for solving real robot tasks.

Considering a more suitable approach for solving real robot task, we discuss point-based POMDP algorithms. These algorithms are capable of solving large POMDP problems by computing an approximate solution and have been applied to several com-

plex robotics tasks including localisation, grasping, navigation, target tracking and exploration. These algorithms have made impressive progress in the field and in some cases, have allowed POMDP problems with hundreds of states to be solved. However, although POMDP planning algorithms have become more scalable and efficient over the years, using such methods in real-world domains can still be challenging.

Pushing further in this direction, we discuss the SARSOP algorithm, an algorithm belonging to the class of point-based POMDP approaches. By focusing on optimally reachable belief spaces, it is able to greatly improve computational efficiency which in turn accelerates computing speeds. SARSOP is an anytime algorithm meaning that the algorithm an be stopped at any time and will always return a solution. However, the quality of the solution improves as more computation time is given to SARSOP. SARSOP computes a good solution much faster than exact POMDP solvers.

However, SARSOP heuristically grows its belief set as it iterates and is difficult to know whether one should not expect any further improvements with regards to its approximation of the optimal value function. The issue is that SARSOP could suddenly discover a new reachable belief with a high expected reward which would then radically change the value function approximation. Because SARSOP maintains both an upper bound and lower bound on the optimal value function, it can stop computations once the bounds converge to within a given gap size of each other. The concern is that for larger domains, the gap size between the upper and lower bound closes very gradually and remains considerable even after the lower bounds seems to have converged [Poupart et al. 2011]. As a result, we leverage the transfer of knowledge from previous tasks, in the form of action priors, to improve the learning rates and expensiveness of solving POMDP tasks with SARSOP.

The SARSOP algorithm is the focus of our study and it is through this algorithm that we hope to show the advantages of the use of action priors. Appropriately so, we then conceptualise the idea of action priors and present the methods of how they can be learned. However, this chapter only illustrates how they can be learned in the MDP case. The methods we use to learn action priors in a POMDP setting can be seen in Section 3.3.1.

Chapter 3

3 Research Method

3.1 Introduction

In conducting this research, we hope to serve two primary functions in the context of reasoning under perceptual uncertainty. The first is to extend previous work on action priors by incorporating them into the machinery of a POMDP solver to demonstrate their benefits in a setting where an agent is unable to directly determine the state of the world in which it lies. Secondly, we aim to improve the performance of a POMDP solver (SARSOP) by drawing on the idea of action regularity. Therefore, this chapter is concerned with the methods and techniques of how we aim to achieve these goals.

We begin by discussing the motivation behind this work in Section 3.2. In the same section, we then present our research hypothesis along with the details that led to our hypothesis. Following our research motivation and hypothesis, Section 3.3 gives an overview of how we plan to tackle the proposed research and plan to validate the accuracy of our presented hypothesis.

3.2 Research Motivation and Hypothesis

Autonomous robots that navigate in real and complex environments are overwhelmed with contingencies and are rarely, if ever, able to access the true state of the world or system [Lankenau and Meyer 1999][Thorpe and Durrant-Whyte 2001][Verma 2005][Kelly et al. 2006]. This causes the robot to become uncertain about its system state. As a result, this uncertainty creates ambiguity in the robot's self-perceived state during the execution of a task.

It is due to this uncertainty that a robot would consider a spot of mud on one of its sensors for an obstacle in the environment in which it is required to navigate. Consequently, its misinterpretation can lead it to make inappropriate decisions such as trying to avoid the obstacle when really such an obstacle does not exist.

Take subterranean spaces such as mines, tunnels, caves and sewers for example. In these environments, a robot may for example have its sensing capabilities impaired due to poor lighting or the presence of dust and/or smoke. Therefore, a major concern for autonomous robots is how to handle situations that are unknown or unanticipated. This phenomenon is known as *operational uncertainty* and can cause unsafe or risky behaviour.

There are two major causes from which uncertainty in terms of system operation may originate. These two sources are namely, the understanding and representation

of the robot's own state and state of the external world, and the limitation of system resources and hence on the representation of the world [Raol and Gopal 2012]. Because the sensors of an agent cannot perceive all the parameters of the external world in which it lies to the highest degree of reliability and because uncertainty may arise as a result of insufficient understandable interpretation of the sensory data it receives, these factors can cause the agent to be uncertain about its current state of the world or system. Thus, from a robot's perspective the uncertainty that we are imperatively required to manage is the uncertainty over the current state of the world and system.

POMDPs provide a principled mathematical framework for planning in uncertain and dynamic environments [Sondik, 1971][Kaelbling, 1998][Png, 2011]. However, in a POMDP, calculating the optimal policy is not easy. This is largely due to fact that POMDPs are characterised with the *curse of dimensionality* and *curse of history* (see Section 2.3.3). Although a number of algorithms have been developed over the years that successfully produce optimal policy solutions in a POMDP context, the downside is that these techniques are too complex and computationally inefficient [Papadimitriou and Tsitsiklis 1987].

Exact POMDP solvers are computationally intractable when they are implemented for solving tasks in real environments. This is because some tasks can contain up to or even more than 10^5 states which can cause their computational complexity to be expensive.

The SARSOP algorithm is an approximate POMDP algorithm that improves computational efficiency and is practical for many applications in robotics [Kurniawati et al. 2008]. SARSOP exploits the idea of optimally reachable beliefs to bring about these computational improvements [Kurniawati et al. 2008]. SARSOP does a great job for reasoning under uncertainty, but it quickly becomes intractable as the number of actions grows because the performance of the algorithm primarily depends on the number of available actions and the number of α-vectors that define the value function. Because the complexity of planning with robots that solve decision problems under uncertainty is compounded as the number of actions available to the agent grows, we look at action priors as a way of controlling this growth.

Action prior knowledge has the ability to improve the performance of learning tasks as they are able to specify action usefulness and guide exploration towards behaviours that have been useful in previously solved tasks. When a robot is faced with a very difficult task, the sensible thing to do would be to allow it to reuse behavioural knowledge acquired from previously solved tasks, rather than allowing it solve the task without any idea over which of its actions are more suitable for particular situations. Allowing it to reuse behavioural knowledge would essentially reduce the amount of time required for it to learn.

Deciding which actions are useful in a situation can be determined by using past experiences of optimal behaviours from multiple previously solved tasks [Rosman and Ramamoorthy 2012]. Because there is some underlying structure common to all these learned tasks as they exist in the same or similar environment, it is this structure we hope to take advantage of to facilitate faster learning. Hence, we are concerned with

allowing agents to be able to learn domain invariances which act as a common sense knowledge of the domain in which it operates. By forming better abstractions of the domain and in turn learning domain invariances that provide insights to elements that are common to a large class of behaviours, this knowledge can offer an understanding in learning which behaviours should be avoided in particular situations.

Consider some learning agent, such as a robot, operating in a maze like-tunnel for a prolonged period of time. In this setting, it may be required to learn to perform various tasks such as, having to navigate from one end of the tunnel to another or from one part of the tunnel to another. In order to accelerate this learning via knowledge transfer, it is clear that the agent would need to be capable of reusing knowledge from other experienced tasks when faced with a new task. However, a key challenge here is how to manage this acquired experience so as to form a mechanism of generalisation, organising it in a manner in which the robot's past behaviours are not specific to previously learned tasks. Because there is some underlying structure common to all these learned tasks as they exist in the same environment, it is this structure we hope to take advantage of to facilitate faster learning.

This research aims to address the problem of operating under uncertainty and keeping this computationally efficient through the use of action priors. Because reliable performance is key to the operational success of a robot, we want to show how action priors can accelerate the learning of policies so that an agent experiencing uncertainty can solve tasks more quickly.

As a solution to a robot having to operate under uncertainty, SARSOP alleviates this issue and may be implemented to help a robot choose actions appropriately. Because we know that action priors are a product of experience and give prior knowledge over the usefulness of actions, we hypothesise that using action priors on a mobile autonomous robot, coupled with the underlying implemented algorithm (SARSOP), will yield greater performance benefits with respect to solving tasks more rapidly. Consequently, our hypothesis is as follows:

- An agent can solve repeated (or multiple) navigation tasks in a setting where it is uncertain about the current state of the world or system faster through the acquisition and use of local domain specific behaviour models. We hypothesise that action priors can be advantageous in a POMDP context (SARSOP) and their use can lead to the computation of good policies in a shorter period of time. In other words, we suggest that an agent acting in a setting where it is uncertain about the current state of the world or system can benefit through the use of having a prior knowledge over actions indicating the preference of the agent in taking particular actions in certain situations. As a result, using an agent's experience of past behaviours can lead to convergence speed improvements.

To help validate our research hypothesis we have formulated six research questions. These are:

1. Can action priors learned from previously solved tasks accelerate the learning of new tasks?

2. Can action prior knowledge learned in one domain be used to accelerate the learning of new tasks in a different domain?

3. Can action priors guide the exploration process in POMDPs away from risky or unsafe behaviours?

4. Does an improved sensing capability improve an agent's ability to leverage action prior knowledge in learning to solve new tasks?

5. Is there a trade-off between the generality of the observations and the usefulness of the perception-based action priors?

6. Can action priors improve the convergence speeds of POMDP algorithms?

However, the primary over-arching question we address in this thesis is thus *can the reuse of behavioural knowledge learned from previously solved tasks accelerate the learning of new tasks?*

In Section 3.3 we propose a method that takes into account the statistics of action choices over a lifetime of an agent, in an effort to allow an agent to be able to quickly cut down options when deciding which actions to select, in a manner which may not have been obvious if each task was solved in isolation. In Section 3.3.1 we present the methods of how we learn action priors, while in Section 3.3.2 we show how action priors can be incorporated into the SARSOP algorithm with the goal of showing how an agent can reuse behavioural knowledge learned from previously solved tasks to accelerate the learning of new tasks.

3.3 Overview of the Research Method

By considering perception-based action priors, an agent is equipped with a mechanism that allows it to be able to perform fast look-ups from any known or unknown location, to determine its preference in choosing actions and can therefore help in controlling the computational explosion of reasoning through chains of actions. Consequently, the information gathered from these priors has the effect of greatly increasing the speed of solving new tasks [Rosman and Ramamoorthy 2012][Rosman and Ramamoorthy 2015]. As a result of their ability to address the limitation of SARSOP in handling situations with a high branching factor (large number of actions) and speed up the learning of new tasks, we incorporate the use of perception-based action priors into the SARSOP algorithm to accelerate SARSOPs learning process.

We present three approaches to learning with action priors in a POMDP context, called SARSOP with Action Priors for Sampling, SARSOP with Action Priors for Pruning and SARSOP with Action Priors for Simulations. During sampling, action priors will be used to guide the search process towards solutions that are most likely to be useful. Hence, they are used to help make action selections so as to bias and efficiently constrain the search for good actions. With regards to pruning, action priors will be used to prune away certain actions using the past experience of optimal behaviours

from many different tasks. During simulations, the action priors will be used to guide the action selection process because in early stages of the learning process, SARSOP can be seen to make bad choices in choosing actions as the learning time has not been sufficient in allowing it to learn the best actions to take in certain situations. By considering the statistics of action choices over an agent's lifetime, the agent will be able to prune less useful actions which will be beneficial to SARSOP's complexity as the computational cost of each iteration depends on the number of available actions. Essentially, this corresponds to pruning away α-vectors based on what were sensible action choices in the past. This is ideal because the set of α-vectors grows exponentially with every algorithm iteration and so keeping this set smaller in size can improve the computational cost of each iteration.

With our evaluation techniques (see Section 3.3.4), we aim to characterise the performance of learning to solve tasks with and without action priors. Our goal is to comparatively illustrate the benefits of learning with action priors to prune certain actions or provide preference in selecting actions as reasoning over a wide branching factor of different action sequences can be very computationally demanding. We are interested in understanding which of these approaches is more rewarding than the others, while also evaluating which of these ideas is more computationally efficient.

This section describes how we conducted the evaluation of the performance of the algorithms we have set out to consider. These algorithms are namely, SARSOP, SARSOP with Action Priors for Sampling, SARSOP with Action Priors for Pruning and SARSOP with Action Priors for Simulations. In Section 3.3.1, we first review the two algorithmic methods for acquiring perception-based action priors, which we will refer to as MDP Action Priors and POMDP Action Priors. Second, we present the three SARSOP approaches for learning with action priors in Section 3.3.2. Section 3.3.3 then discusses the design of the experiments, and lastly, in Section 3.3.4, we outline the performance measure used to compare the various algorithms presented in Section 3.3.2.

3.3.1 Learning Action Priors

In this section, we present the algorithms we will use to obtain context dependent prior knowledge over an agent's actions. These action priors are conditioned on the observations of the agent. We consider the MDP as well as the POMDP case for the learning of action priors. The approaches to how an agent will learn this prior knowledge over actions will vary depending on the paradigm that is being considered.

Learning Action Priors for Markov Decision Processes

When an agent learns action priors in a MDP, we refer to these priors as MDP Action Priors. In order to learn these, an agent is required to solve various tasks in an environment with fully observable state information. By solving multiple tasks in the same domain and studying the optimal policies that arise from those tasks, we hope to learn

about the structure of the underlying domain. In other words, we use the set of optimal Q-functions to extract the action priors as a form of structural information about the domain (see Section 2.6.2).

Consider a learning agent operating in a domain $D = (S, A, T, \gamma)$ that is required to perform multiple tasks in this domain. We define a task by the tuple $M = (D, R)$ as the MDP where the state set, action set, observation set, transition function and observation function are fixed for the entire domain. Considering the set of tasks $\mathcal{M} = \{M\}$ the agent is required to perform and the set of optimal policies $\prod = \{\pi_P\}$ corresponding to those tasks in $\mathcal{M}$, learning the action prior involves learning a distribution $\theta_o^{\mathcal{M}}(a)$ over the action set for each observation $o \in \mathcal{O}$.

The prior $\theta_o^{\mathcal{M}}(a)$ represents the action prior learned in a MDP context and is a bias over the action set of the agent. The prior $\theta_o^{\mathcal{M}}(a)$ gives the probability of each action $a \in A$ being used by an optimal policy when observation o is observed, averaged over tasks. We assume here that different tasks differ only in the reward function and $\theta_o^{\mathcal{M}}(a)$ symbolises the preference of the agent in taking some action given an observation.

Consider two actions, a_1 and a_2. We say action a_1 is more useful than action a_2 when observation o is observed, if $\theta_o^{\mathcal{M}}(a_1) > \theta_o^{\mathcal{M}}(a_2)$. This implies that a_1 is used by more optimal policies than a_2 when the observation is o. As a result, this information can provide the agent with knowledge about which actions are useful in situations (which are differentiated by its observation) where the agent has several choices to explore. This is particularly useful for seeding search in a policy learning process. Thus, Action priors can help the agent avoid harmful actions during the execution of task.

To learn the action priors, we allow an agent to solve a set of tasks and use the computed Q-functions $Q(s, a)$ of those tasks to acquire knowledge of the structural information of the domain. Note that $Q(s, a)$ is computed using Q-learning. The approach we use to learn the action priors from a set of tasks in a MDP context is shown in Algorithm 7.

In order to model the action priors $\theta_o^{\mathcal{M}}(a)$, for each observation o, we maintain a count $\alpha_o^{\mathcal{M}}(a)$ for each action. The count $\alpha_o^{\mathcal{M}}(a)$ represents the α_o counts obtained in a MDP context. The α_o counts can be initialised to any value so as to portray some prior knowledge. However, after a new Q-function $Q(s, a)$ for some task is learned, the α_o counts are updated according to $Q(s, a)$ (see Algorithm 7 line 15). In this way, $\alpha_o^{\mathcal{M}}(a)$ tells us how many times action a was considered the best action to take (i.e. the action that lead to the highest return) when observation o was observed in any Q-function. Essentially, $\alpha_o^{\mathcal{M}}(a)$ corresponds to the number of times action a was considered the optimal action when the agent's observation was o. Note that the learning rate $\alpha^Q \in [0, 1]$ in Algorithm 7 should not be confused with the α_o counts $\alpha_o^{\mathcal{M}}(a)$.

The action priors $\theta_o^{\mathcal{M}}(a)$ are modeled using a Dirichlet distribution. We obtain $\theta_o^{\mathcal{M}}(a)$ by sampling from the Dirichlet distribution $\theta_o^{\mathcal{M}}(a) \sim \text{Dir}(\alpha_o^{\mathcal{M}})$ where $\sum_a \theta_o^{\mathcal{M}}(a) = 1, \forall o \in \mathcal{O}$. If $\alpha_o^{\mathcal{M}}(a)$ is equivalent for each action given an agent's observation i.e. $\alpha_o^{\mathcal{M}}(a) = c, \forall a \in A$, the resulting prior $\theta_o^{\mathcal{M}}(a)$ can be seen to be uniform, implying that each action is equally favourable given that observation.

One of the strengths of learning MDP action priors, using the method presented

in Algorithm 7, is that when a new Q-functions is learned and used to update $\alpha_o^{\mathcal{M}}$, more than one action could be used to update $\alpha_o^{\mathcal{M}}$ for a given observation. This is because the Q-function can have other actions which have a value equal to that of the optimal action at a given state and will consider all these actions when updating $\alpha_o^{\mathcal{M}}$. This has the effect of learning a prior that can generalise well to new tasks. However, learning action priors using Q-functions can be time consuming as each new Q-function can take a considerable amount of time to converge. This is because during the learning process, Q-learning explores actions uniformly and converges only when every state in the domain is visited sufficiently often and $Q(s, a)$ value updates between trials are small. Exploration functions ensure that unfamiliar state-action pairs are recognised and are a fix for situations where one part of the state space has been sufficiently explored and other parts have not [Klein and Abbeel 2013]. However, we do not consider exploration functions in this thesis.

Algorithm 7 uses ε-greedy Q-learning as the underlying algorithm and is designed to output a distribution $\theta_o^{\mathcal{M}}(a)$ over the action set, representing the probability of each action being used by an optimal policy relative to a particular observation. In Section 3.3.2 we give a discussion of how the learned priors $\theta_o^{\mathcal{M}}(a)$ can be used in practice and present an algorithm that achieves this (see Algorithm 9, 10, 11).

Algorithm 7 Learning Perception-based Action Priors in a MDP.

Initialise $\alpha_o^{\mathcal{M}}(a)$ arbitrarily
Initialise $Q(s, a)$ arbitrarily
for every task $i = 1 \dots n$ **do**
    for every episode $k = 1 \dots K$ **do**
        Choose initial state s
        repeat
            Randomly choose action a
            Take action a, observe r, s'
            $Q(s, a) \leftarrow Q(s, a) + \alpha^Q \left[r(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
            $s \leftarrow s'$
        until s is terminal
    end for
    for every state $s = s_1 \dots s_n$ **do**
        $o \leftarrow$ observations(s)
        $\alpha_o^{\mathcal{M}}(a) \leftarrow \begin{cases} \alpha_o^{\mathcal{M}}(a) + 1 & \text{if } a = \arg\max_a Q(s, a) \\ \alpha_o^{\mathcal{M}}(a) & \text{otherwise} \end{cases}$
    end for
end for
$\theta_o^{\mathcal{M}}(a) \sim \mathrm{Dir}(\alpha_o^{\mathcal{M}})$
return $\theta_o^{\mathcal{M}}(a)$

Learning Action Priors for Partially Observable Markov Decision Processes

Contrary to learning action priors in the MDP paradigm, we also present an approach to learning action priors in an environment with partially observable state information. Learning these priors involves solving multiple POMDP tasks. We refer to these as POMDP Action Priors. We learn these by computing approximately optimal value functions for various tasks using SARSOP and then use those value functions in simulations to track the actions that are considered to be useful. This complies with studying the optimal policies that arise from solving tasks in order to acquire prior knowledge over actions.

Consider a learning agent operating in a domain $D = (S, A, \mathcal{O}, T, Z, \gamma)$ that is required to perform multiple tasks in this domain. We define a task by the tuple $P = (D, R)$ as the POMDP where the state set, action set, observation set, transition function and observation function are fixed for the entire domain. Considering the set of tasks $\mathcal{P} = \{P\}$ the agent is required to perform and the set of approximately optimal policies $\prod = \{\pi_P\}$ corresponding to those tasks in $\mathcal{P}$, learning the action prior involves learning a distribution $\theta_o^{\mathcal{P}}(a)$ over the action set for each observation $o \in \mathcal{O}$.

The prior $\theta_o^{\mathcal{P}}(a)$ represents the action prior learned in a POMDP context and is a bias over the action set of the agent. The prior $\theta_o^{\mathcal{P}}(a)$ gives the probability of each action $a \in A$ being used by an approximately optimal policy when observation o is observed, averaged over tasks. We assume here that different tasks differ only in the reward function and $\theta_o^{\mathcal{P}}(a)$ symbolises the preference of the agent in taking some action given an observation.

Consider two actions, a_1 and a_2. We say action a_1 is more useful than action a_2 when observation o is observed, if $\theta_o^{\mathcal{P}}(a_1) > \theta_o^{\mathcal{P}}(a_2)$. This implies that a_1 is used by more approximately optimal policies than a_2 when the observation is o. Similarly to the MDP case, this information can provide the agent with knowledge about which actions are useful in situations (which are differentiated by its observation) where the agent has several choices to explore. This is particularly useful for seeding search in a policy learning process. Thus, the use of action priors can help the agent avoid harmful actions during the execution of a task.

To learn the action priors, we allow an agent to solve a set of tasks and use the computed approximately optimal value functions $V(b)$ of those tasks to acquire knowledge of the structural information of the domain. Note that to compute $V(b)$ in this thesis, we use the SARSOP algorithm. However, the ideas presented in this thesis are not specific to SARSOP. Our approach to learning the action priors from a set of tasks is shown in Algorithm 8.

In order to model the action priors $\theta_o^{\mathcal{P}}(a)$, for each observation o, we maintain a count $\alpha_o^{\mathcal{P}}(a)$ for each action. The count $\alpha_o^{\mathcal{P}}(a)$ represents the α_o counts obtained in a POMDP context. As in the MDP case, the α_o counts can be initialised to any value so as to portray some prior knowledge. However, after an approximately optimal value function $V(b)$ for some task is learned, the α_o counts are updated according to $V(b)$ during the simulation of that task (see Algorithm 8 lines 9). Essentially, $\alpha_o^{\mathcal{P}}(a)$

corresponds to the number of times action a was considered the optimal action when the agent's observation was o.

The action priors $\theta_o^{\mathcal{P}}(a)$ are modeled using a Dirichlet distribution. We obtain $\theta_o^{\mathcal{P}}(a)$ by sampling from the Dirichlet distribution $\theta_o^{\mathcal{P}}(a) \sim \text{Dir}(\alpha_o^{\mathcal{P}})$ where $\sum_a \theta_o^{\mathcal{P}}(a) = 1, \forall o \in \mathcal{O}$. If $\alpha_o^{\mathcal{P}}(a)$ is equivalent for each action given an agent's observation i.e. $\alpha_o^{\mathcal{P}}(a) = c, \forall a \in A$, the resulting prior $\theta_o^{\mathcal{P}}(a)$ can be seen to be uniform, implying that each action is equally favourable given that observation.

The difference between learning the POMDP action priors $\theta_o^{\mathcal{P}}(a)$ and MDP action priors $\theta_o^{\mathcal{M}}(a)$ is that when we learn $\theta_o^{\mathcal{P}}(a)$ a POMDP framework is used and when we learn $\theta_o^{\mathcal{M}}(a)$ an MDP framework is used. This means learning $\theta_o^{\mathcal{P}}(a)$ occurs over the belief space $\mathcal{B}$, while learning $\theta_o^{\mathcal{M}}(a)$ occurs over the state space S. The effect is that when we update the α_o counts, we use an approximately optimal value function $V(b)$ computed by SARSOP in the POMDP context and use a computed optimal Q-function $Q(s, a)$ in the MDP context. These differences can be seen by referring to Algorithm 8 line 9 and Algorithm 7 line 15. In both instances, we have assumed that the world, which is potentially infinite in states and infinite in beliefs, is composed of a finite number of observational patterns.

An advantage of learning POMDP action priors, using the method presented in Algorithm 8, is that once an approximately optimal value function is computed, a large number of updates to $\alpha_o^{\mathcal{P}}$ can be made with the value function from one task. This is because we use a simulation step (Algorithm 8 line 4-16) to update $\alpha_o^{\mathcal{P}}$ and a large number of simulations can be used. The number of simulations chosen to be used is up to the user, but we note that if more simulations are used, more knowledge can be obtained producing a more informative action prior. As a result, the amount of knowledge that can be obtained from one task is far greater than that of the method of learning MDP action priors using $Q(s, a)$ (see Algorithm 7). However, the disadvantage is that only one action can be considered when updating $\alpha_o^{\mathcal{P}}$ for a given observation. This is a result of SARSOP's δ-dominance pruning technique (see Section 2.5.3) as it only considers one α-vector to be optimal at a given belief. Therefore only one action will be considered when updating $\alpha_o^{\mathcal{P}}$ for a given observation each time an update is made.

Algorithm 8 uses SARSOP as the underlying algorithm and is designed to output a distribution $\theta_o^{\mathcal{P}}(a)$ over the action set, representing the probability of each action being used by an approximately optimal policy. We omit the details of the SARSOP algorithm in Algorithm 8. However, we encourage the reader to refer to Section 2.5 for the workings and background of this algorithm if required. In Section 3.3.2, we give a discussion of how the learned priors $\theta_o^{\mathcal{P}}(a)$ can be used in practice and present an algorithm that achieves this (see Algorithm 9, 10, 11).

Quality of the Action Priors

Learning action priors requires extensive exploration of the space of possibilities to find invariances in the domain, across policies. These invariances refer to the aspects of the

Algorithm 8 Learning Perception-based Action Priors in a POMDP.

Initialise $\alpha_o^{\mathcal{P}}(a)$ arbitrarily
for every task $i = 1 \dots n$ **do**
  Compute $V(b)$ for task i using SARSOP
  for every simulation $k = 1 \dots K$ **do**
    Set belief b to b_0
    Choose initial state s, observe o
    repeat
      $a \leftarrow \arg\max_{a \in A} V(b)$
      $\alpha_o^{\mathcal{P}}(a) \leftarrow \begin{cases} \alpha_o^{\mathcal{P}}(a) + 1 & \text{if } a \text{ is chosen using } V(b) \\ \alpha_o^{\mathcal{P}}(a) & \text{otherwise} \end{cases}$
      Take action a, observe s', o'
      $b' \leftarrow \dfrac{Z(s', a, o') \sum_{s \in S} T(s, a, s') b(s)}{P(o'|b, a)}$
      $s \leftarrow s'$
      $o \leftarrow o'$
      $b \leftarrow b'$
    until s is terminal
  end for
end for
$\theta_o^{\mathcal{P}}(a) \sim \mathrm{Dir}(\alpha_o^{\mathcal{P}})$
return $\theta_o^{\mathcal{P}}(a)$

domain whereby regardless of the task that is being executed, the agent's interaction with the domain will remain the same. For example, whether a person is participating/running in a 100 metre sprint or 200 metre sprint race, the rules of the race, running technique used and interaction protocols with other runners remain unchanged regardless of the type of race. Domain invariances are useful in a lifelong sense because they have the ability to factor out the elements which remain unchanged across specific tasks. As a result, learning these domain invariances simplify the problem of having to perform a new task. Because different behaviours have commonalities at a local level, we consider these invariances to be the aspects of the domain which an agent would treat the same way, regardless of the task it is required to execute. Thus, by taking action priors into account, we are able to inject prior knowledge learned from previously solved tasks which represents the sensible behaviours in the domain.

The key assumption in this thesis is that in a domain in which and agent is required to perform multiple tasks over a long period of time, there is certain structure in the domain, which is in the form of local contexts, that can result in certain actions to be commonly selected or always avoided as they are either harmful or do not contribute towards completing any task. By learning this structure, local sparsity in the action selection process is induced. Therefore, we regard this as a form of transfer learning [Thrun1996][Caruana 1997], where an agent is required to solve some tasks and learn to generalise the knowledge learned from solving those tasks so that it can be applied to others.

By learning from more tasks, an agent may be able to acquire more information resulting in a more informative prior knowledge. This is because an agent is able to more accurately determine the actions that may be commonly selected in situations by having more information from previously learned tasks at its disposal. Therefore, by learning from multiple tasks, an agent may be able to acquire a better quality prior over its actions possibly allowing it to better bias its exploratory behaviours. As a result, we gauge the quality of the learned action priors based on the number of tasks it has learned to solve.

In Section 4.2, we show how the quality of the action prior knowledge increases with the number of learning tasks. These results can be observed in Figure 17, 24 and 36. Now that we have discussed how action priors can be learned, we need to see how they can be used in practice.

3.3.2 SARSOP with Action Priors

We propose that for an agent to be generally capable when it is required to perform a range of unknown tasks in which it experiences uncertainty, the agent must have the ability to continually learn from a lifetime of experience which is largely dependent on two abilities. Firstly, is its ability to generalise from past experiences and secondly is its ability to form representations which facilitate faster learning and the transfer of knowledge between different situations. By exploiting the commonalities between large families of tasks, the agent may potentially minimise situations where it has to relearn

from scratch. This also has the effect of allowing an agent to build better models of the domain. Therefore, we hope to facilitate faster learning by learning such regularities from the domain and extracting the common elements between tasks.

Consider some learning agent that has prolonged experience in a domain. This means the agent has learned to solve multiple tasks in the domain. However, with each task it learned to solve in the domain, it would have had to relearn everything about the domain for every new task. This results in a slow learning process. The main problem we address in this section is how we can accelerate the learning of new tasks through the use of transferring knowledge learned from previously solved tasks.

Using action priors, we can provide an agent with knowledge over which actions may be sensible to take in particular circumstances. By incorporating action priors in the learning process of an agent, the agent can bias its exploratory behaviour based on actions that have been useful in the past in similar situations, but different tasks. Essentially, they provide information over the usefulness of each action in situations where the agent has a variety of action choices to explore. For this reason, they can be useful for seeding search as well as pruning and prioritising actions in a policy learning process This has the effect of improving the performance of learning tasks in a domain.

By considering action priors, an agent is equipped with a mechanism that allows it to be able to perform fast look-ups from any known or unknown situation, to determine its preference in choosing actions. Therefore action priors can help in controlling the computational explosion of reasoning through large chains of actions. The knowledge gathered from these priors may have the effect of greatly increasing the speed of solving new tasks. As a result of their ability to address the limitation of SARSOP in handling situations with a high branching factor (large number of actions) and their ability to speed up the learning of new tasks, we show how we can incorporate the use of action priors into the SARSOP algorithm to accelerate SARSOP's learning process.

We demonstrate the advantages of using action priors in a POMDP context with an adaptation of the traditional SARSOP algorithm. We present three algorithms for policy learning with action priors, called SARSOP with Action Priors for Sampling (Algorithm 9), SARSOP with Action Priors for Pruning (Algorithm 10) and SARSOP with Action Priors for Simulations (Algorithm 11). Note that the methods we use to use action priors in SARSOP is not specific to SARSOP and are more broadly applicable to other similar decision making approaches.

SARSOP with Action Priors for Sampling

In order to make good action choices there is a need to evaluate the future effects of taking those actions, so as to approximate the value of each choice. However, this is an expensive search process. This is because the number of possible future outcomes are exponential in the number of action choices to be made.

Alternatively, by implementing action priors we are able to prune and prioritise the available actions based on past experiences. Effectively, action priors allow us to prune poor action choices based on previously solved tasks, thus reducing the complexity of

lookahead search.

Using action priors in the sampling stage of SARSOP (see Section 2.5.2) is the most intuitive and is based on the original way in which action priors were used in Q-learning (see Section 2.6). By using action priors in the action selection step (Algorithm 9 line 18), we bias the exploration of actions away from less useful actions. By taking into account the statistics of action choices over a lifetime of the agent, we can quickly cut down options when deciding which actions to select, in a manner which may not have been obvious when each task is solved in isolation. Thus, by taking the actions that were considered to be useful in past situations, we can potentially ensure that beliefs with high expected rewards are reached by the SARSOP algorithm. This could have the effect of ensuring that sampling is focused on the optimally reachable belief space $\mathcal{R}^*$. Therefore, in this way SARSOP may be able to form better approximations of the optimal value function in a shorter period of time.

The difference between the SARSOP with Action Priors for Sampling algorithm and the traditional SARSOP algorithm can be seen on line 18 of Algorithm 9. The action selection step consists of two cases. The first case deals with exploring actions by choosing the action with the highest upper bound with probability $1 - \varepsilon$, while the second case deals with exploring actions by choosing actions based on the prior $\theta_o(a)$ with probability ε. Choosing actions based on the prior $\theta_o(a)$ allows us to shape the action selection based on what were sensible actions in the past. In essence, we want the agent to be able to also explore the actions that were favoured in previously learned tasks.

The downside of using action priors in this manner is owed to the fact that we select actions based on the prior $\theta_o(a)$. If sufficient learning time is not given to learning the action priors or the priors do not generalise well, there may be instances where the action priors may lead SARSOP down a sampling path outside of $\mathcal{R}^*$. As a result, the computed optimal value function approximation could potentially cause the agent to learn suboptimal policies. However, on the upside, using action priors allows the agent to make fast lookups when selecting which action to take during sampling. In contrast, SARSOP has to search through the entire set of α-vectors, that define the current approximation of the value function, to find the α-vector with the highest upperbound value and then select the action with which the α-vector is associated.

Because action priors prune and prioritise action selection in a context dependent manner, this method of using action priors can be easily applicable to other point-based POMDP solvers. This is possible because all such approaches sample beliefs by taking an action and making an observation from the belief space using the belief update (Equation 14-19) in a similar fashion to SARSOP.

In Algorithm 8, we do not present the details of the backup and pruning strategies as they are the same as those of the traditional SARSOP approach. However, for additional details refer to Section 2.5.1.

Algorithm 9 SARSOP with Action Priors for Sampling.

Initialise the lower bound $\underline{V}$ on the optimal value function V^* and the upper bound $\overline{V}$ on V^*.
Insert the initial belief point b_0 as the root of the tree $T_{\mathcal{R}}$.
Initialise action priors $\theta_o(a)$
Initialise observation o
while termination condition is not reached **do**
    SAMPLE($T_{\mathcal{R}}$, Γ).
    Select a subset of nodes from $T_{\mathcal{R}}$ and perform BACKUP($T_{\mathcal{R}}$, Γ, b) on each selected node b.
    PRUNE($T_{\mathcal{R}}$, Γ).
return Γ.

SAMPLE($T_{\mathcal{R}}$, Γ)
    Set L to the current lower bound on the value function at the root b_0 of tree $T_{\mathcal{R}}$ and set U to $L + \Phi$, where Φ is the target gap size at the root.
    SAMPLEPOINTS($T_{\mathcal{R}}$, Γ, b_0, L, U, Φ, 1)

SAMPLEPOINTS($T_{\mathcal{R}}$, Γ, b, L, U, Φ, h)
    if $\hat{V} \leq L$ and $\overline{V}(b) \leq \max\left\{U, \underline{V}(b) + \Phi\gamma^{-h}\right\}$, where $\hat{V}$ is the prediction of $V^*(b)$ **then**
        return
    else
        $\underline{Q} \leftarrow \max_a \underline{Q}(b, a)$
        $L' \leftarrow \max\left\{L, \underline{Q}\right\}$
        $U' \leftarrow \max\left\{U, \underline{Q} + \Phi\gamma^{-h}\right\}$
        $a' \leftarrow \begin{cases} \arg\max_a \overline{Q}(b, a) & \text{w.p. } 1 - \varepsilon \\ a \in A & \text{w.p. } \varepsilon\theta_o(a) \end{cases}$
        $o' \leftarrow \arg\max_o P(o|b, a') \left[\overline{V}(\tau(b, a', o)) - \underline{V}(\tau(b, a', o))\right]$
        $L_h \leftarrow \dfrac{L' - \sum_{s \in S} R(s, a')b(s) - \gamma \sum_{o \neq o'} P(o|b, a')\underline{V}(\tau(b, a', o))}{\gamma P(o'|b, a')}$
        $U_h \leftarrow \dfrac{U' - \sum_{s \in S} R(s, a')b(s) - \gamma \sum_{o \neq o'} P(o|b, a')\overline{V}(\tau(b, a', o))}{\gamma P(o'|b, a')}$
        $b' \leftarrow \tau(b, a', o')$
        Insert b' into $T_{\mathcal{R}}$ as a child of b
        SAMPLEPOINTS($T_{\mathcal{R}}$, Γ, b', L_h, U_h, Φ, $h + 1$)

SARSOP with Action Priors for Pruning

Through the use of action priors and their ability to prune and prioritise action choices in a context dependent manner, they can provide substantial benefits to learning and decision making [Rosman and Ramamoorthy 2012][Rosman 2014][Rosman and Ramamoorthy 2015]. Thus, we incorporate action priors into the SARSOP algorithm to create a more aggressive pruning approach. The performance of the SARSOP algorithm largely depends on the number of α-vectors in the set Γ representing the value function (see Section 2.3.4). Therefore, by implementing action priors so as to further prune the actions that are seemingly suboptimal, we decrease the set Γ and ultimately improve the performance of the overall algorithm.

Algorithm 10 lines 10-12, presents an additional pruning strategy designed to prune alpha vectors based on action priors. Given an observation o, we choose the action to be pruned using $1-\theta_o(a)$. However, we only prune that action only if the prior $\theta_o(a)$ is smaller than some threshold $\xi \in [0,1]$. By using ξ, we are able to control the α-vectors that are pruned so as to ensure that useful α-vectors are kept in the set Γ while less useful α-vectors are pruned away (see Figure 18). This will allow SARSOP to take advantage of both pruning using δ-dominance as well as pruning using action priors.

An area of concern for using action priors in this manner is that it adds an aggressive pruning strategy. This is riskier as pruning occurs over the observational space $\mathcal{O}$ which has a finite number of observational patterns rather than over a subset of the belief space B which can be considerably larger than the observational space. This additional pruning could possibly cause the value function approximation to degrade. However, if we have action priors that accurately model the entire action distribution over all tasks, then the priors should be able to prune α-vectors appropriately and in turn result in faster learning times.

An important property of point-based algorithms is the ability to prune α-vectors that are dominated by others in B. However, these pruning methods are generally computationally intensive, and there is currently no fast method for pruning α-vectors [Cassandra et al. 1997][Guy et al. 2013]. As such our method of pruning can be used as an additional pruning technique in other point-based POMDP solvers as an attempt to accelerate the learning process. Thus, using action priors for pruning may generalise to other point-based algorithms that are not SARSOP.

The sample, backup and prune methods in Algorithm 10 are those of the SARSOP algorithm and for this reason, we are not concerned with their workings in this section. However, these methods are discussed in detail in Section 2.5.

SARSOP with Action Priors for Simulations

SARSOP consists of two phases, namely the learning phase and the simulation phase. The learning phase consists of computing an approximately optimal value function, while the simulation phase consists of using the computed value function from the learning phase to execute a task. Action priors can be used during the action selection step of the simulation phase of SARSOP (see Algorithm 11 line 8). There are two cases

Algorithm 10 SARSOP with Action Priors for Pruning.

1: Initialise the lower bound $\underline{V}$ on the optimal value function V^* and the upper bound $\overline{V}$ on V^*.
2: Insert the initial belief point b_0 as the root of the tree $T_{\mathcal{R}}$.
3: Initialise action priors $\theta_o(a)$
4: **while** termination condition is not reached **do**
5: SAMPLE($T_{\mathcal{R}}$, Γ).
6: Select a subset of nodes from $T_{\mathcal{R}}$ and perform BACKUP($T_{\mathcal{R}}$, Γ, b) on each selected node b.
7: PRUNE($T_{\mathcal{R}}$, Γ).
8: PRUNEWITHACTIONPRIORS(Γ, o).
9: **return** Γ.

PRUNEWITHACTIONPRIORS(Γ, o).
10: choose action a w.p. $1-\theta_o(a)$
11: **if** $\theta_o(a) < \xi$ **then**
12: prune all α-vectors of observation o in Γ relative to a

in deciding which action to take during this action selection step. The action selected in this step is determined by either exploiting the current policy given by $V(b)$ or by choosing the action based on the prior $\theta_o(a)$. The case involving action prior usage is handled by selecting the action from A with probability based on $\theta_o(a)$, with each action chosen proportional to the number of times that action was considered a sensible choice in the past. Traditional SARSOP chooses actions based only on the current estimate of $V(b)$, so in the early stages of the learning process, where the estimate of $V(b)$ is poor, SARSOP can be seen to select less useful actions. Therefore, we show that using action priors in this way can bias exploration away from less useful actions by restricting search over the action space.

The parameter $\varepsilon \in [0, 1]$ controls the trade-off between exploiting the current policy and using the action priors to select the action to be taken. Note that the parameter ε is annealed after each sampling iteration. This means the agent is able to exploit its current estimate of the optimal policy, while also incorporating a prior knowledge over actions allowing it to choose actions based on previously successful behaviours.

During simulations, in early stages of the learning process, SARSOP can be seen to make bad choices in choosing actions as it has not been given sufficient learning time so that it is able to learn the best actions to take at certain beliefs. Therefore, we use action priors to guide and prioritise the action selection process away from actions that are less useful based on previously solved tasks. As a result the action priors may be able to accelerate the learning of tasks. Primarily, the difference between our approach of selecting actions during the simulation phase and that of SARSOP is that we choose actions based on the prior $\theta_o(a)$, while SARSOP selects the action associated with the α-vector that has the highest value $V(b)$ at belief b. Thus, although SARSOP

Algorithm 11 SARSOP with Action Priors for Simulations.

Initialise $\alpha_o(a)$ arbitrarily
Compute $V(b)$ for task P using SARSOP
for every simulation $k = 1 \dots K$ **do**
  Set belief b to b_0
  Choose initial state s, observe o
  repeat
    $a \leftarrow \begin{cases} a = \arg\max_{a \in A} V(b) & \text{w.p. } 1 - \varepsilon \\ a \in A & \text{w.p. } \varepsilon \theta_o(a) \end{cases}$
    Take action a, observe s', o'
    $b' \leftarrow \dfrac{Z(s', a, o') \sum_{s \in S} T(s, a, s') b(s)}{P(o'|b, a)}$
    $s \leftarrow s'$
    $o \leftarrow o'$
    $b \leftarrow b'$
  until s is terminal
end for

may choose the action associated with the α-vector that has the highest value at the current belief, it may not be the best action to take as the learning time may have been inadequate in enabling it to be able to differentiate between useful, detrimental or useless actions. Therefore, we use action priors to improve the decision making in this context.

A particular issue that may arise from using action priors in this manner is that, although the action priors may guide the action selection process towards useful actions, the agent may struggle to ultimately solve its task by reaching its goal. This issue may be caused by the fact that we choose actions based on the prior $\theta_o(a)$. For example, consider an agent having to navigate in a building from one location in a room to a location in another room. The action priors may be able to safely and successfully guide the agent to the destination room, but finding the exact location to which it should navigate to in that room may be difficult. However, if we have action priors that accurately model the entire action distribution over all tasks, instances where this issue may arise should be minimised.

Point-based algorithms generally include a learning phase and a simulation phase which involves taking a sequence of actions until a goal state is reached. Therefore, we can use the methods we developed for using action priors during the simulation phase of SARSOP on other point-based algorithms to guide their action selection process.

3.3.3 Experiment Design

The structure of the experiments are as follows. We consider four spatial navigation domains to demonstrate the benefits of using action priors as we believe that spatial navigation is a setting in which action priors stand to make significant gains. We consider a maze domain (see Section 4.2.1), a lattice domain (see Section 4.2.2) which is a variant of the maze domain, a light-dark domain (see Section 4.2.3), and a hallway domain (see Section 4.2.4). Each domain environment is modeled using grid cells and the agent's task is to travel from some initial location to a goal location while attempting to avoid obstacles. The agent can travel in four possible directions, namely North, South, West and East. These directions make up the agent's action set and taking each action involves moving one cell in the intended direction. The agent's actions in the maze domain, lattice domain, and hallway domain are deterministic, while those in the light-dark domain are non-deterministic. The reward structure is defined as follows: the agent's arrival at the goal state will yield the agent a reward of 100, a wall collision a reward of -10, while each action taken will result in a reward of -1.

An agent is given a navigation objective in which it is required to navigate from some initial position to a goal location. The agent has 8 sensors, which allow it to perceive the occupancy of the 8 cells surrounding its current location. The agent operates under a set of assumptions which state that sensors do not fail and are errorless, all objects observed by the agent are obstacles, all actions taken by the agent will either cause it to move to a new state or remain in its current state (depending on if the target cell is empty) and any prior knowledge the agent is given is true.

Prior to task execution, the agent will have a prior knowledge over actions in the form of action priors (see Section 2.6). The agent will be uncertain about the state in which it lies during task execution, therefore the agent will be required to make observations to gather information in an attempt to determine its current state. We note that the agent's prior knowledge will be acquired in a smaller, but similar world to the world in which algorithm performance comparisons will be made. Hence, the goal of the experiments is to observe whether an agent having some form of experience will be able to obtain richer policies in a shorter period of time.

In learning action priors, a smaller world domain is used in comparison to the domain that is used for testing purposes. However, the domain that is used for learning action priors will share similar characteristics with its larger successor. This similarity is achieved by ensuring that the same observational patterns that the larger domain is composed of make up the smaller domain as well. The tasks the agent will be required to complete in the smaller domain will therefore be similar to those used for testing. We want the agent to be able to form better abstractions of the domain through which it will be required to navigate during testing by solving tasks in the smaller domain and using the knowledge it learns from solving those tasks.

We use 20 random tasks, with each task consisting of 100000 episodes, to train the action priors. Note that each algorithm will use the same tasks and simulations will consist of 1000 episodes. In an episode, an agent is initialised to a random starting

state and required to travel to its goal state. Episodes are terminated when the agent reaches its goal location or after a 1000 simulation steps have been taken.

In each episode run of a simulation, the discounted reward is collected. We accumulate these discounted rewards for all episodes and average over the total number of episodes. This gives an approximation of the total expected discounted score that the agent would receive for following a particular value function approximation. The total expected discounted reward will be calculated for all the tasks given for the agent to solve and averaged over these tasks. In each experiment, we use this as our basis on which to compare the performance results of each algorithm.

3.3.4 Experiment Performance Metrics

When we consider planning for POMDP tasks, the goal is typically to maximise the expected accumulation of discounted rewards. Computing this expectation exactly, requires us to examine all possible action-observation trajectories with a length equal to the planning horizon. However, because there are as many possible observations after taking each action as there are observations, the number of such trajectories grows exponentially with the horizon. Therefore, because computing the exact expectation in this manner is not feasible, this method is not used in general.

Instead, by simulating sampled trials in the environment, we can alternatively approximate the computation of the exact expectation. In each trial, the agent begins at an initial belief point and executes actions by following the value function computed by SARSOP. By averaging over multiple executions, we compute the average discounted reward which is essentially an unbiased estimator of the expectation. The average discounted reward is computed to be

$$Ave.\ Reward = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=0}^{k}\gamma^{j}r_{j} \tag{52}$$

where n is the number of trials and k is the trial length.

As a display of our results, we present the policy qualities with respect to learning times given a range of SARSOP sampling iterations. The average discounted reward is reported after a particular number of sampling iterations have elapsed, while also recording the time in which it took to run through the specific number of iterations. This allows us to evaluate the rate at which the different algorithms converge so that we can observe which approach gives good solutions more quickly.

To compare the performance and efficiency of the algorithms, we use four performance measures. We consider the average discounted reward versus the number of sampling iterations, time versus the number of sampling iterations, percentage of tasks completed versus number of sampling iterations and the average number of steps taken to complete tasks versus the number of sampling iterations. Note that in SARSOP a sampling iteration refers to sampling a belief point which is obtained by taking an action at a belief and making an observation. Although comparing the reward received with

respect to time may be problematic when comparing results over different computers, we note that it is required to assess the extent to which action priors improve computing times. Thus, we believe that comparing the reward received versus the learning time is a reasonable approach for the purposes of this analysis.

3.4 Conclusion

We hypothesise that an agent can solve repeated (or multiple) navigation tasks in a setting where it is uncertain about the current state of the world or system faster through the acquisition and use of local domain specific behaviour models. In particular, we hope to show that action priors can be advantageous in a POMDP context (SARSOP) and their use can lead to the computation of good policies in a shorter period of time.

As such, we consider the SARSOP algorithm, a point-based POMDP approach, as the basis of our research work. By identifying the complexity of solving POMDPs as a major issue, our goal is to illustrate the benefits of incorporating prior knowledge over actions in an uncertain environment. We consider uncertainty in robot control and sensor measurements to show the advantages of using action priors in this setting.

We develop three new algorithms called SARSOP with Action Priors for Sampling, SARSOP with Action Priors for Pruning and SARSOP with Action Priors for Simulations to show the advantages of using action priors in a setting where an agent experiences uncertainty. However, to acquire the prior knowledge over actions to be used in these algorithms, an agent learns action priors using two different approaches. We allow an agent to learn action priors in an MDP and POMDP setting. Note that each action prior learning technique has advantages and disadvantages associated with its use (see Section 3.3.1).

We also present the design of our experiments (see Section 3.3.3) as well as the performance metrics which we use with the experiments presented in Section 4.2. Using these performance metrics, we aim to characterise the performance of learning to solve tasks with and without action priors. Our goal is to comparatively illustrate the benefits of learning with action priors to prune certain actions or provide preference in selecting actions as reasoning over a wide branching factor of different action sequences can be very computationally demanding.

Chapter 4

4 Results and Analysis

4.1 Introduction

This section presents a set of simulated experiments designed to evaluate the computed policies resulting from the traditional SARSOP algorithm and the SARSOP with action priors algorithms. In particular, the aim of these experiments is to observe whether incorporating action priors in the SARSOP algorithm will accelerate the computation of an approximately optimal value function, while also increasing the expected reward when solving particular tasks.

In this section, we also present the results to the experiments described herein and give an in-depth analysis of those results. We begin by presenting the results that illustrate the varying performances of the traditional SARSOP approach to the SARSOP with Action Priors approaches, coupled with an analysis of the results. Our focus is to understand the usefulness of action priors in a POMDP context and to evaluate whether action priors indeed reduce computation times and improve solution quality so as to improve algorithm performance.

We hypothesise that action priors can be advantageous in a POMDP context (SARSOP) and their use can lead to the computation of good policies in a shorter period of time. Therefore, with these experiments we hope to show that an agent acting in a setting where it is uncertain about the current state of the world or system can benefit through the use of having a prior knowledge over actions, which indicates the preference of the agent in taking particular actions in certain situations. As a result, using an agent's experience of past behaviours should lead to convergence speed improvements.

4.2 Experiments

The experimental procedure involves two phases. The first phase is concerned with generating a set of tasks in a domain in order for the agent to learn the corresponding estimates of the optimal value functions, so that the action priors $\theta_o^{\mathcal{M}}(a)$ and $\theta_o^{\mathcal{P}}(a)$ can be learned from them, using the methods presented in Algorithm 7 and 8. The second phase is concerned with generating a new set of tasks for the agent to complete and using the learned action priors to do so, using the methods presented in Algorithm 9, 10 and 11. Note that the results in this section do not include the time required to learn action priors. This is because the learning of action priors is a once-off requirement which can be incorporated into the learning process of any future task.

For each experiment, we applied each SARSOP with Action Priors algorithm to a number of different tasks. For each task, we ran a sufficiently large number of simulation runs to approximately determine the expected total reward of the resulting policy (see Section 3.3.3 details of the experiment design). For comparison purposes, we also ran the traditional SARSOP algorithm on the same set of tasks. For each algorithm, the expected total reward for each task was used to produce an average reward. All considered algorithms were implemented using C++. The experiments were performed on a PC with a 3.40GHz Intel Processor and 8GB RAM.

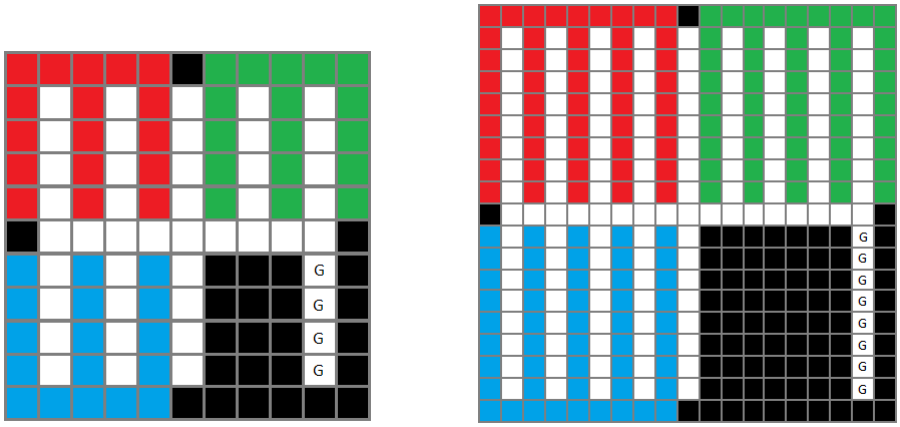
4.2.1 Maze Domain

In Figure 15, we present a simple maze domain designed to evaluate our proposed methods. Primarily the experiments run in this domain are to show which SARSOP with action prior algorithm(s) potentially has the ability to improve the performance of learning to solve POMDP tasks.

The maze domain simulates an environment where an agent has limited sensing capabilities. The various grid cells of colour represent wall obstacles of different classes, while the white cells represent passable space. All white grid cells not marked with a "G" represent possible initial positions, uniformly selected at random. All cells marked with a "G" represent possible goal states and each task can be differentiated by the position of the goal. The agent's actions are deterministic and a task involves starting at an initial position and taking a sequence of actions until the goal is reached. Note that with each episode of a task, the agent is initialised to some random location.

The maze domain that is used for our experiments is presented in Figure 15. The domain in Figure 15a is a smaller, similar domain to the domain presented in Figure 15b. The domain in Figure 15a is used to learn the action priors, while the domain in Figure 15b is used to test the learning of tasks with action priors. Note the set of observational patterns in both domains are the same. We learn action priors in a smaller domain not only to show that action prior knowledge learned for a set of observations is transferable to domains with different state spaces, but also because we can learn action priors more rapidly in a less complex domain. This is necessary because the computations of the approximately optimal value $V(b)$ and Q-function $Q(s, a)$ used to learn the priors can take a considerable amount of time to reach convergence.

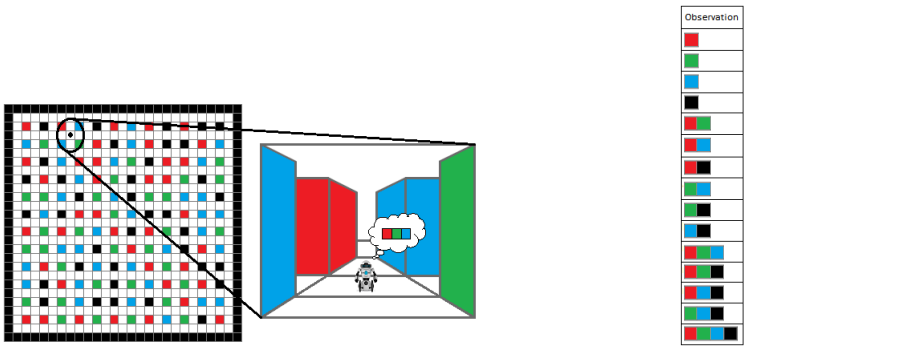
We structure this domain in this manner so that an agent operating in such an environment will not be able to determine its state accurately at any instance. The wall colour configuration of the domain is designed in a manner to bring about sufficient ambiguity with regards to the agent's ability to perceive and predict the state in which it lies. The maze domain has been carefully structured, ensuring that it is not overly ambiguous so that navigation tasks can in fact be completed by the agent. The goal was to create a domain that is ambiguous enough to cause the agent to experience sufficient uncertainty while performing the tasks, and still making it possible for the agent to complete those tasks within that domain.



a) Maze domain used to learn action priors. b) Maze domain used for testing action prior use.

Figure 15: The maze domain.

We simulate a scaled down sonar by allowing the agent to be able to perceive only the colour information of the eight grid cells that surround its current cell location (see Figure 16). Using these eight grid cells, the agent will be able to gather information over its current position in space and where it may lie in its world. Note that because the agent is operating under uncertainty, its current position is never completely known to the agent. To ensure that this sensory information is ambiguous when the agent interprets the surrounding grid cells, it will not be capable of determining the wall to which a particular colour belongs. All that it is able to conclude is the colour, but will not be able to determine which sensor detected it. In addition, the agent is also not capable of determining the number of times a particular wall colour occurs when perceiving its surrounding grid cells. For example, if the agent perceives two blue wall obstacles, a green wall obstacle and a red wall obstacle surrounding its current grid cell location, it will only register that it observed a blue, green and red wall obstacle, ignoring the number of times a particular wall colour appeared in its observation. The only information the agent has at its disposal with certainty are the wall colours that surround it. This example is portrayed in Figure 16.



a) Agent's observation. b) List of possible types of observations.

Figure 16: Agent's perception capability in maze domain.

Because we have two methods with which to learn action priors, one using the MDP paradigm and another using the POMDP paradigm (see Algorithm 7 and 8), we are also concerned with determining the approach that works best with point-based solvers. Figure 17 shows that the action priors learned from using the POMDP paradigm approach (Algorithm 8) produce better results than the priors learned from using the MDP paradigm approach (Algorithm 7). We can also observe that the POMDP paradigm approach reaches convergence far more quickly as well. This is because when we learn POMDP action priors $\theta_o^{\mathcal{P}}(a)$, more information can be learned from a policy learned from one task. This is a result of the simulation step of Algorithm 8 (see Algorithm 8 line 4 -16). Because a large number of simulations is used, 100000 to be exact, there is a large number of updates to the α_o-counts $\alpha_o^{\mathcal{P}}$ (see Algorithm 8 line 9) resulting in more knowledge being learned and ultimately resulting in a more informative prior using a smaller number of training tasks.

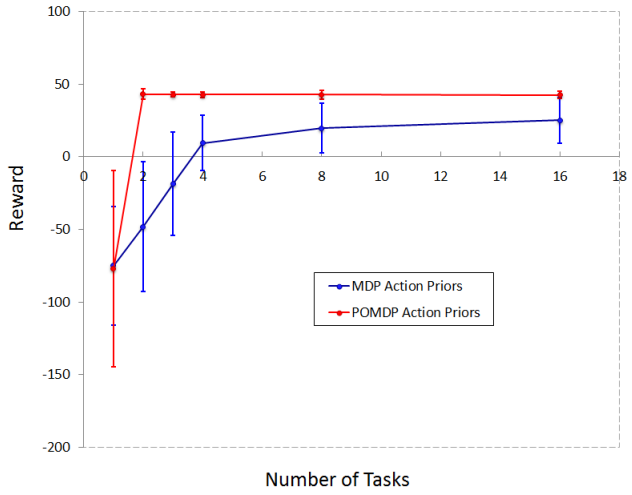


Figure 17: Action prior learning technique results in maze domain. The results are averaged over 16 tasks.

We are also concerned with determining which action prior use approach yields the best results. The algorithm that yields the best results will therefore then be used in subsequent experiments. However, before we can run this comparison experiment, we must first discover the optimal threshold value $\xi \in [0, 1]$ (see Algorithm 10 line 11) for the SARSOP with Action Priors for Pruning algorithm (Algorithm 10). Note that ξ is the only free parameter of the proposed methods. Once we discover this optimal ξ value, we are then able to produce the optimal form of the SARSOP with Action Priors for Pruning algorithm for this domain.

Figure 18 shows the results of using various ξ values and we can observe that using values from a range of 0.1 to 0.9 yield very similar results. Note the action priors used in this experiment were POMDP action priors $\theta_o^{\mathcal{P}}(a)$ as they achieved the best results in our previous experiment (see Figure 17). Similar results illustrated by Figure 18 are owed to the fact the probability distributions of $\theta_o^{\mathcal{P}}(a)$ for each action at a given observation is so heavily weighted towards one particular action that changing the threshold value ξ does not produce a major difference between the performance curves presented in Figure 18. By studying Figure 15a, we are able to notice that every other observation pattern, except the observation used for a possible goal state (black observation), has one useful action associated with it. A useful action in this context implies an action preference that will lead the agent to reaching a particular goal. As a result, $\theta_o^{\mathcal{P}}(a)$ produces a heavy weighting towards one particular action at a given observation. As a result, we choose the value of ξ to be 0.1 in our following experiment.

We note that these results presented in Figure 18 are domain specific and changing ξ when considering a different domain may yield different results. However, we do not consider this approach in other experiments due to the results presented in Figure 19 which tell us that SARSOP with Action Priors for Pruning (Algorithm 10), is not the best of the three approaches we consider in this thesis.

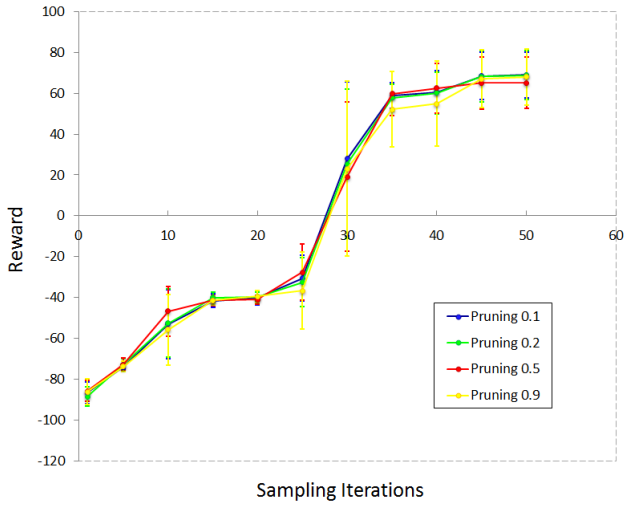


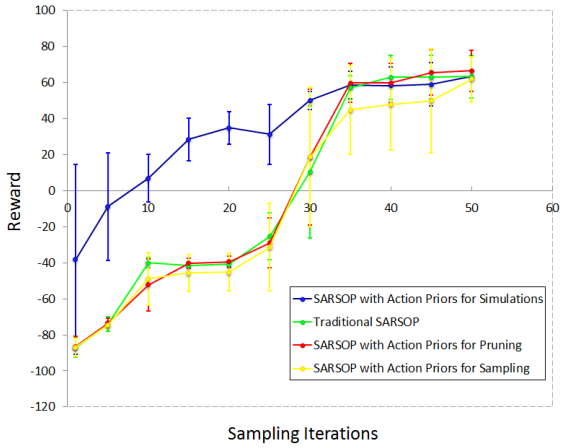
Figure 18: Threshold experiment for SARSOP with Action Priors for Pruning. The results are averaged over 16 tasks.

Figure 19 shows the performance curves of the three different approaches to using action priors along with the traditional SARSOP algorithm. These three different approaches are presented in Algorithm 9, 10 and 11. Note that in Algorithm 9 and 11, the use of action priors is annealed after each sampling iteration. This has the effect of ensuring that the agent is able to exploit its current estimate of the optimal policy, while also incorporating a prior knowledge over actions allowing it to choose actions based on previously successful behaviours. Essentially, this allows action priors to be used more often at early stages of the learning process and less later in the learning process. This means even at early stages of the learning process, the agent is able to avoid detrimental actions such as colliding into walls and results in higher returns even after a few sampling iterations during learning.

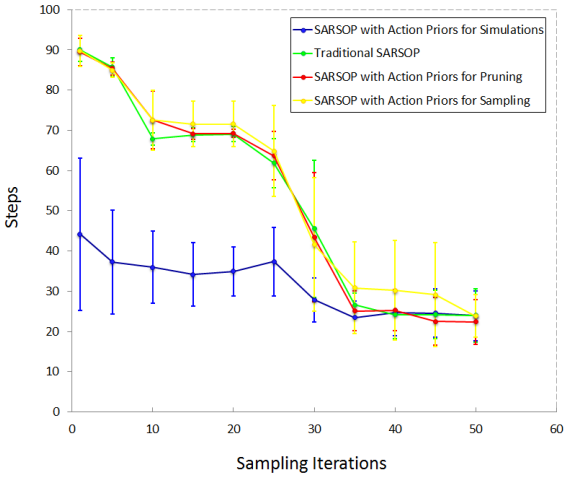
Observing the results illustrated in Figure 19, we can conclude that the best of the three methods of using action priors with SARSOP is by incorporating action priors in the simulation phase. SARSOP with Action Priors for Pruning (Algorithm 10) produces similar results to the traditional SARSOP because it can only do as well as the traditional SARSOP as the optimal α-vectors in Γ of the traditional SARSOP will be the same as those in SARSOP with Action Priors for Pruning. However, because the set Γ in SARSOP with Action Priors for Pruning will be smaller than that of the traditional SARSOP as a result of the additional pruning, this suggests that SARSOP with Action Priors for Pruning may yield improvements in terms of algorithm convergence speeds (see Section 2.3.4 and 2.4.2). This is an experiment we carry out later in this section (see Figure 30). SARSOP with Action Priors for Sampling performs worse than the traditional SARSOP because it turns out that we can only guarantee convergence by choosing the action with the highest upper bound value. By choosing a suboptimal action, we eventually discover its suboptimality when the upper-bound of that action drops below the upper bound of another action.

Each algorithm has an issue that may affect its performance. Because we base the selection of actions on the prior $\theta_o(a)$ during sampling, there are instances where SARSOP with Action Priors for Sampling samples beliefs outside the optimally reachable belief space $\mathcal{R}^*$. Traditional SARSOP ensures that we sample beliefs near $\mathcal{R}^*$ by taking the action with the highest upper bound value and so when this is not consistently maintained, we may sample beliefs outside of $\mathcal{R}^*$. On the other hand, because we incorporate an additional pruning strategy into SARSOP when we use SARSOP with Action Priors for Pruning, in some instances, this additional pruning strategy could possibly cause the value function approximation to degrade as a result of over pruning. Lastly, one particular issue that may arise from using SARSOP with Action Priors for Simulations is that although the action priors successfully guide the agent to areas of interest in the domain, the agent may struggle to find the goal once it arrives at these areas i.e. it may successfully reach the area of the maze where it is surrounded by black walls but struggle to find the exact location of the goal (see Figure 15b).

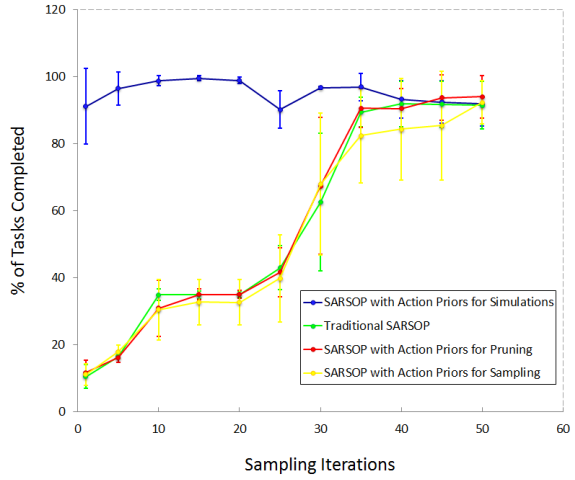
Figure 19 shows the policy improvements that are associated with using action priors in the simulation phase of SARSOP. SARSOP with Action Priors for Simulations illustrates that even from the onset of the learning process using action priors result in policies that yield a positive return, solve tasks quicker and allow an agent to solve a large percentage of tasks that it is given. Because SARSOP is not able to differentiate between useful, detrimental or useless actions at early stages of the learning process as a result of inadequate learning time, using action priors at these stages allows SARSOP to be equipped with a mechanism that allows it to determine the actions that are useful when perceiving a given observation. Thus, action priors provide the agent with knowledge over which actions to avoid as they may either be harmful or useless i.e. may cause the agent to collide with wall obstacles or may lead the agent astray causing it to land up in locations further away from the goal.



a)



b)



c)

Figure 19: Action prior use approach results. The results are averaged over 16 tasks.

The results in Figure 20 compare the performance per sampling iteration of a learning agent using a set of correct action priors and incorrect action priors. Using correct action priors with SARSOP illustrates that the priors allow SARSOP to reduce the cost of the early stages of the learning process.

The correct action priors are learned from solving multiple tasks, and guide the agent towards areas where the agent would be surrounded by black walls. From the onset of learning, the correct action priors allow the agent to have some "common sense" knowledge including not moving into walls, moving to areas close to the goal location etc. On the other hand, the incorrect priors are established by inverting the knowledge learned from learning the correct priors. This means where the correct action prior would suggest to take the action to move South, the incorrect prior would indicate a preference to take the action to move North and where the correct prior would suggest to move East, the incorrect prior would indicate a preference to take the action to move West and vice versa.

The results in Figure 20 reinforce the idea that the incorrect priors are a bad knowledge base as well as an example of negative transfer and can be observed with the terrible performance in comparison to the traditional SARSOP algorithm. This makes the additional point that using a different distribution for selecting actions can have a massive effect on policy performance - this itself is a new contribution and something that has not been explored before in this setting. It is clear you can do far more harm with this than good, and that seems to make it a dangerous approach. However, it is clear that learning the action priors in this way from previously learned tasks actually helps performance.

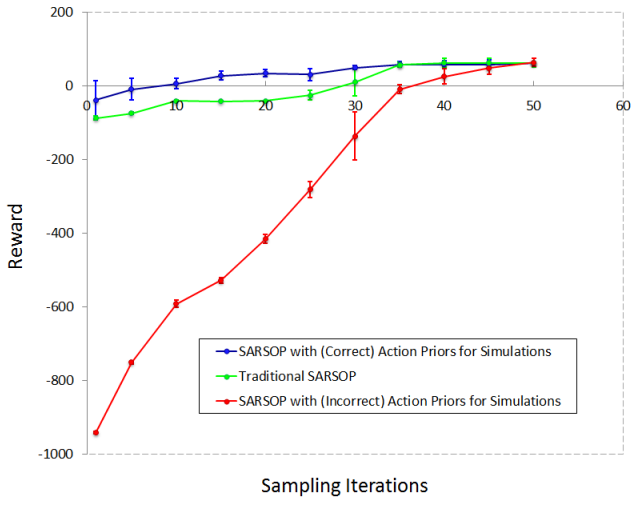


Figure 20: Correct vs incorrect action priors. The correct action priors are learned from solving multiple tasks, while the incorrect priors are established by inverting the knowledge learned from learning the correct priors. The results are averaged over 16 tasks.

As another example of negative transfer, we show how learning action priors using the domain given in Figure 15a and used in Figure 21 can be detrimental to the learning of tasks. Note that the colour of the goal region in Figure 21 is blue and not black as it was in the domain used to learn the action priors (Figure 15a). As a result, the action priors have a negative effect on the performance of SARSOP with Action Priors for Simulations (see Figure 22). This is the case because when the priors are learned in the domain given in Figure 15a, the agent will learn that when a blue colour is observed the most useful action is to move North. However, observing Figure 21 we can observe that taking the action to move North when the agent is surrounded by blue walls is not the only useful action in solving most tasks. Another issue is that the observations the agent makes at location A and B in Figure 21 are never experienced when learning action priors in the domain given in Figure 15a. Therefore the action priors do not provide the agent with knowledge on how to treat these situations (i.e. do not allow the agent to be able to determine the most useful action under that observation) and chooses the action to move North, South, West and East uniformly which can result in taking useless or detrimental actions as the priors can lead the agent in the wrong direction or colliding into walls. This also emphasises the importance of learning action priors in a domain that allows the agent to learn sufficient domain knowledge that can generalise to other similar domains.

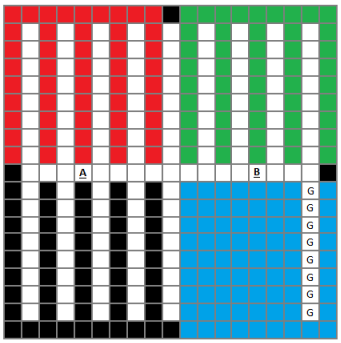


Figure 21: Maze domain with goal region colour change.

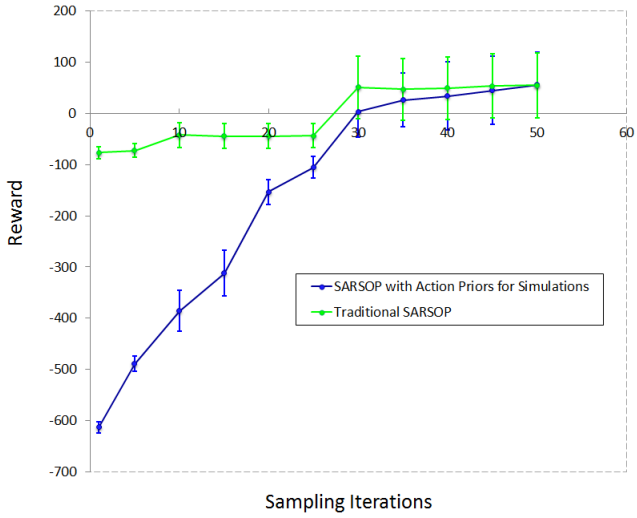
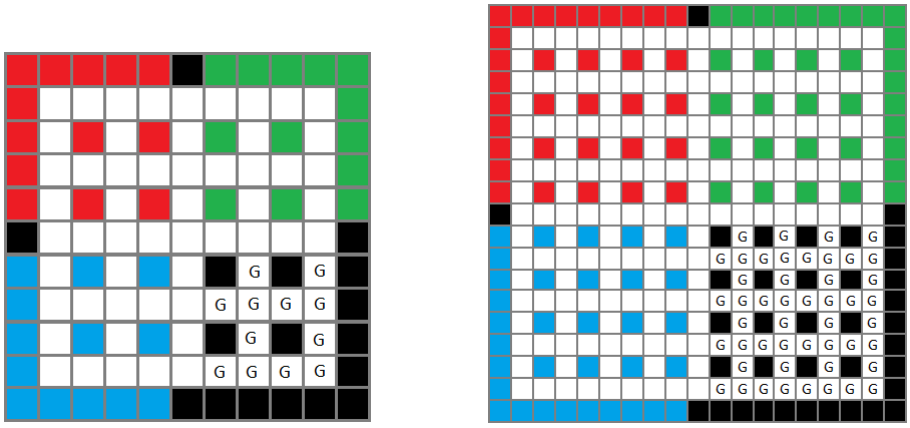


Figure 22: Incorrect usage of action priors. This involves taking the action priors learned from the domain presented in Figure 15a and using them to solve tasks in the domain presented in Figure 21. The results are averaged over 16 tasks.

4.2.2 Lattice Domain

The lattice domain, presented in Figure 23, is a variant of the maze domain presented in Figure 15. Again, we use a smaller, similar domain to learn action priors to that used for testing the action priors to reinforce the idea that action prior knowledge can be transferred to domains with different state spaces and transition functions provided that the perceptual contexts of the agent remain the same across trials. We also note that the value function $V(b)$ and Q-function $Q(s, a)$ used to obtain action prior knowledge are computed faster in a smaller domain. Here, the lattice domain is introduced so that we can explore cases where our approach may induce negative transfer.

The agent's navigational space is enclosed with walls with every second row and column of the grid representing passable space. The remaining cells are used to represent pillar like wall obstacles which result in a lattice like domain (see Figure 23). Once again, all white grid cells not marked with a "G" represent possible initial positions, uniformly selected at random, while all cells marked with a "G" represent possible goal states with tasks differing from one another based on the position of the goal. Completing a particular task includes navigating from the initial position and reaching the specified goal location. Here, a similar sensing capability to that presented in Figure 16 is employed to represent the agent's observations.



a) Lattice domain used to learn action priors. b) Lattice domain used for testing action prior use.

Figure 23: The lattice domain.

As in the previous experiment, we start off by testing which action prior learning technique, between Algorithm 7 and 8, yields the best results when incorporated into SARSOP. Figure 24, displays the results of the action prior learning techniques and as in the previous experiment (see Figure 17), the POMDP paradigm approach (Algorithm 8) produces a higher return with each learned task and ultimately represents better knowledge to be used to accelerate the learning of future tasks. As a result, these will be the priors that will be used in subsequent experiments relating to this lattice domain.

Note that the gap between the performance curves, in Figure 24, of the two methods of learning action priors in this experiment is much smaller than that of the previous experiment (see Figure 17). This is due to a combination of two factors. Firstly, the observation signals are too sparse and do not provide the agent with sufficient knowledge so that it can make safer action selections (we discuss this in more detail later in this section). Therefore, even though the POMDP action priors can obtain more knowledge from tasks as a result of the large number of simulations during learning (see Section 3.3.1), the knowledge it obtains does not produce the best priors. Secondly, when a new Q-function is learned and used to update the α_o-counts $\alpha_o^{\mathcal{M}}$ to learn the MDP action priors, more than one action can be used to update $\alpha_o^{\mathcal{M}}$ for a given observation. This is because the Q-function can have two actions which have a value equal to that of the optimal action at a given state and will consider both these actions when updating $\alpha_o^{\mathcal{M}}$. A combination of these two factors helps close the gap between the performance curves of the two action prior learning methods.

To help describe this more clearly, consider location C in Figure 26 for example. After learning the POMDP action priors you can observe that when the agent is in the red area of the maze, it has a higher preference to taking the action to move South and a lower preference to taking the action to move East. This means there is a higher probability that the agent will collide with the wall obstacles in locations similar to C. However, after learning the MDP action priors you can observe that when the agent is in the red area of the maze, it has equal preference to taking the action to move

South and East. Therefore, the probability of colliding into the wall at locations like C is lower when using MDP action priors as a result of considering two actions when updating $\alpha_o^{\mathcal{M}}$. On the contrary though, because the POMDP action priors are able to gain more knowledge from a smaller number of tasks, they are still able to outperform MDP action priors.

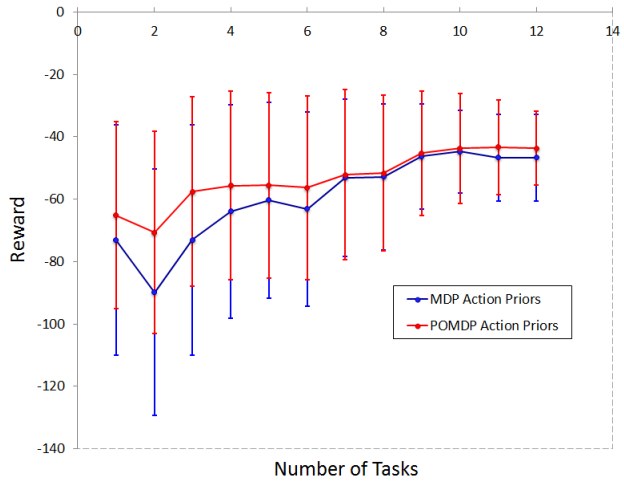


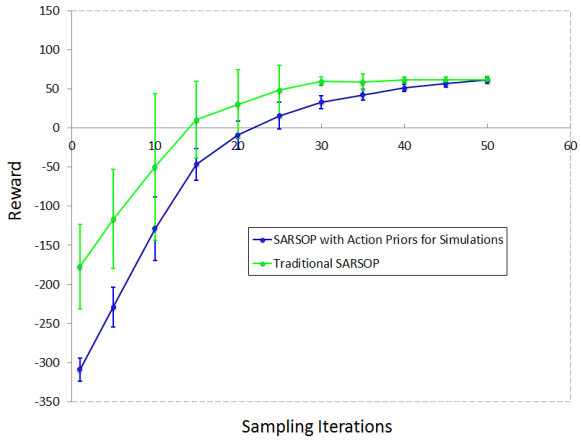
Figure 24: Action prior learning technique results in lattice domain. The results are averaged over 48 tasks.

The results of Figure 25 show that the priors learned and used in this experiment negatively affect the performance of SARSOP as the traditional SARSOP algorithm outperforms SARSOP with Action Priors for Simulations. This is because the observation signals of the agent are too sparse. In other words, the observations of the agent are too ambiguous and do not provide the agent with sufficient information about its surroundings. Thus, in some cases the action priors will lead the agent to taking harmful or useless actions as the priors are learned over the observation space of an agent. Note that this is called negative transfer [Taylor and Stone 2009].

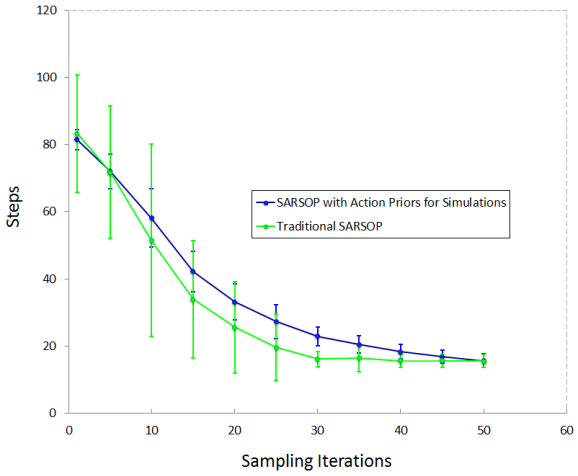
Take for example position A and B in Figure 26. Although the agent's observation will be the same in both locations, as the agent will only be able to determine that it is in the red part of the lattice domain, the best actions to take at these 2 locations are different. The action priors suggest that taking the action to move South or East are considered to be optimal in this area, and either choice is a feasible action to take. However, while either action is a reasonable choice at location A, this is not the case at location B. Taking the action to move East at location B or at a similar location will result in the agent colliding with the wall. This phenomenon occurs in many areas of the lattice domain which causes SARSOP with Action Priors for Simulations to perform poorly in comparison to SARSOP. This result suggests that improving the agent's observation capabilities and learning action priors over these new observation capabilities will improve the performance of the SARSOP with Action Priors for Simulations

algorithm. This idea makes up the work of our next experiment and the results can be seen in Figure 28 and Figure 29.

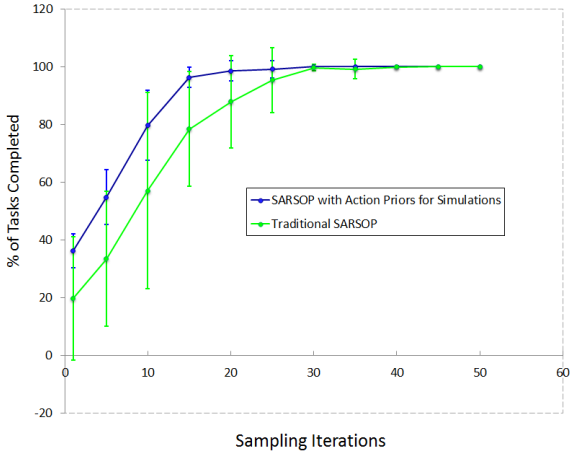
Despite the negative impact the action priors have on the SARSOP algorithm when observations are ambiguous as seen in Figure 25a and 25b, interestingly enough the priors are still able to increase the percentage of tasks that are completed in the early stages of the learning process (see Figure 25c). Thus, although the action priors result in longer and unsafe or risky task executions, they still manage to encourage more tasks to be completed by guiding the agent to areas of interest. This means the action priors will provide a preference to actions that will cause the agent to reach areas of the domain where a goal state is likely to be encountered. In this case, these areas of interest are those that are marked with a "G" (referencing possible goal locations) in Figure 23b.



a)



b)



c)

Figure 25: Performance results in lattice domain. The results are averaged over 48 tasks.

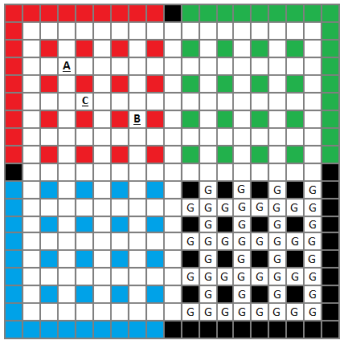


Figure 26: Lattice domain location comparison.

The results in Figure 28 show the performance gains that are associated with an improved sensing capability when using action priors. The improved observations allow the agent to determine not only the colours of the walls that surround it, but also the exact position of walls. We allow the agent to perceive the occupancy of the 8 grid cells that surround the current location of the agent (see Figure 27). Thus, the action priors learned over this improved observation capability allows the agent to prevent wall collisions completely, giving the results of Figure 28. The improved observation capability allows both action prior learning techniques to improve their performance. Once again, since the POMDP action prior learning technique of Algorithm 8 outperforms its MDP counterpart of Algorithm 7, these will be the priors that we use in the rest of the lattice domain experiments that follow.

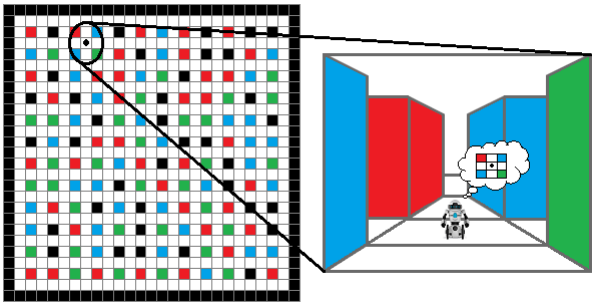


Figure 27: Agent's improved perception capability in lattice domain.

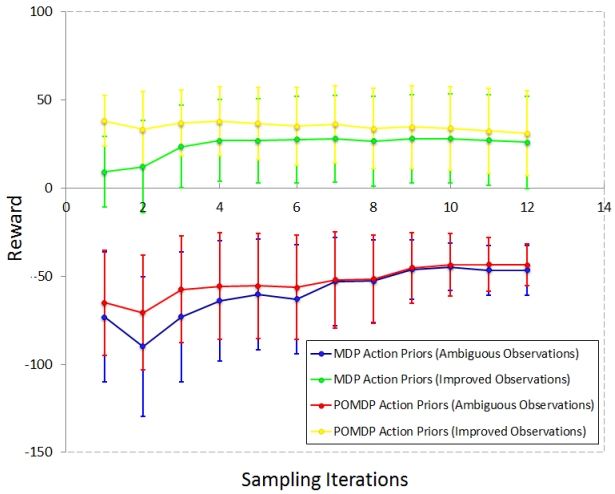
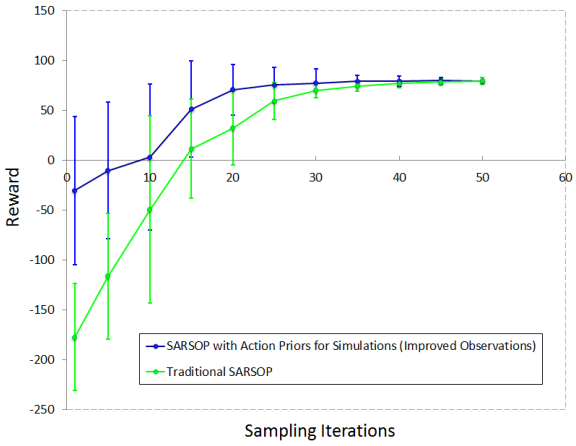


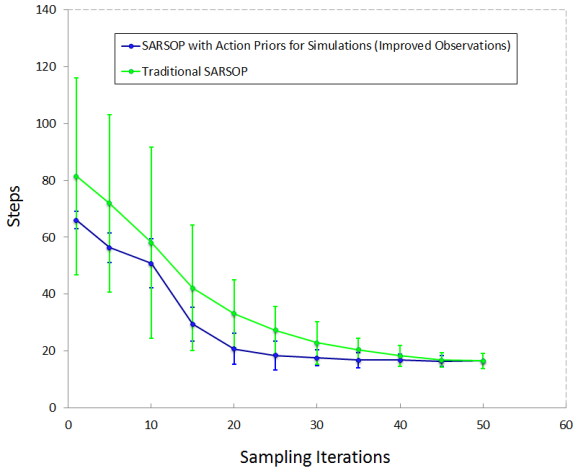
Figure 28: Action prior learning technique results for precise observations vs ambiguous observations. The results are averaged over 48 tasks.

Figure 29 demonstrates the performance of the action priors over the improved sensing capability of the agent. In this case, the agent is able to correctly exploit the number of invariances that arise in the lattice domain. Because the agent is able to precisely locate the position of walls, it alleviates the issue of classifying harmful actions as optimal actions as a result of an ambiguous observation space. This means given location A and B in Figure 26, the improved observation allows the agent to be able to differentiate between the two locations and no longer considers them to be treated in a similar fashion.

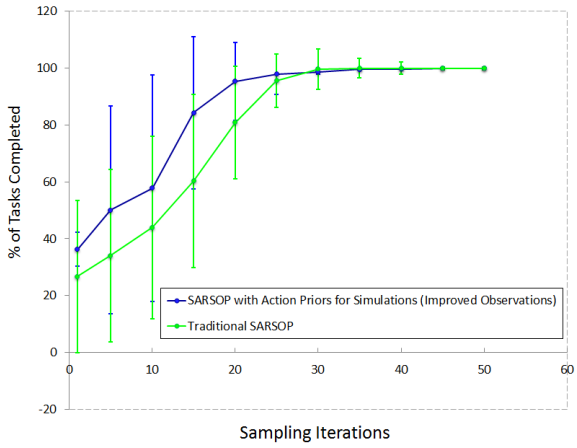
The benefits of action priors rely heavily on the observational features it uses to define an observation when learning the priors. This is because the amount of action information that can be learned depends largely on the observation features it uses when defining an observation. There is clearly a trade-off between the generality of the observations and the usefulness of the action priors. As we have seen with the results in Figure 25 and Figure 29, the more general an observation is defined to be, the usefulness of using the action priors will be lower. In other words, the more general the observation features, the less informative the action priors will be. However, the more specific the observational features are, the less portable the action priors are to new states. Note that the observational features the agent is able to use depend on properties of the task and domain. Generally, action priors are particularly useful in rich environments that are characterised with repeating structure.



a)



b)

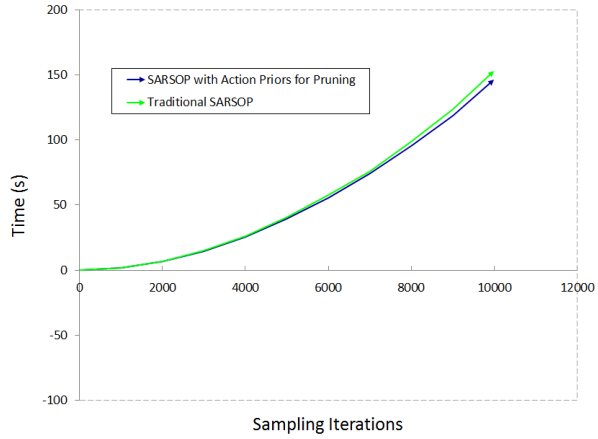


c)

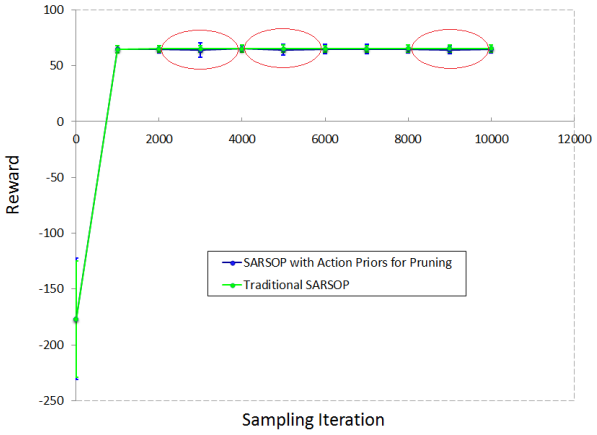
Figure 29: Performance results in lattice domain using an improved observation sensor. The results are averaged over 48 tasks.

Figure 30a illustrates how the SARSOP with Action Priors for Pruning (Algorithm 10) can successfully be used to speed up algorithm convergence. Note that SARSOP performs slightly better than SARSOP with Action Priors for Pruning (see Figure 30b). The red circles in Figure 30b highlight areas where traditional SARSOP's reward is higher than SARSOP with Action Priors for Pruning. Therefore, there is a trade-off between the accuracy of the value function and algorithm computation times. However, the difference in solution quality here is negligible in comparison to the amount of computational time that could potentially be saved. This reinforces the theory and conclusions mentioned in Section 2.4.1.

By considering Figure 30a, we can observe how this additional pruning helps with accelerating algorithm computations as the number of sampling iterations increase. Note how the SARSOP with Action Priors for Pruning and SARSOP start off with similar computation times with no time difference between the two performance curves, but as the sampling iterations increase, we can see this time difference increase showing that the additional pruning is speeding up SARSOP as a result of reasoning over fewer α-vectors. These results can be expected to improve further by considering a greater number of sampling iterations.



a)



b)

Figure 30: Convergence speed results in lattice domain. The results are averaged over 48 tasks.

4.2.3 Light-Dark Domain

The light-dark domain aims to simulate a robot in a simple mine or cave, with poor sensing capabilities. With this domain we want to show that even with stochastic actions, using action priors can still display performance gains with regards to learning to solve tasks faster. Ultimately, our goal is to show that action priors can still be beneficial in guiding an agent's action selection process in a setting where it has poor sensing capabilities and stochastic actions.

The grid cells marked with an "S" represent the possible initial positions for the agent, while the cells marked with a "G" represent possible goal locations (see Figure 31). The grid cells that are red in colour represent wall obstacles, while the black, gray and white cells represent passable space. The varying shades of black, gray and white represent varying intensities of light, with the white cells resembling areas where the light is at its brightest. In this domain, the agent's ability to localise itself depends

on the amount light at its current location. In other words, the darker the cell colour in which the agent resides, the more uncertain the agent is about its current state. Completing a particular task includes navigating from an initial position and reaching the specified goal location.

The action priors we use in the experiments in this section are those learned using Algorithm 8 (POMDP action priors). We use POMDP action priors because based on our previous two experiments (see Section 4.2.1 and 4.2.2) they produced the best action prior knowledge when compared to that of the MDP action priors (Algorithm 7). The action priors are learned in the same domain as the domain used for testing. However, the tasks that are used to learn action priors are different to those that are used to test the usefulness of the priors.

The agent's actions also become more accurate depending on the magnitude of light over the cell in which it lies. For example, there is a higher probability of landing in the left or right cell of the agent's current position when taking the action to move North in a darker area, than when the agent is in an area with more light (see Figure 32). An observation is taken to be the grid cell colour of the agent's current position. Note the agent operates under uncertainty in this domain and the true location is not known. Therefore, the agent is required to make observations to approximate its location in space.

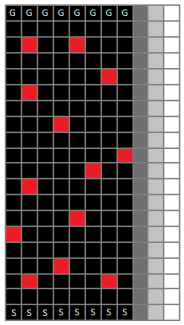
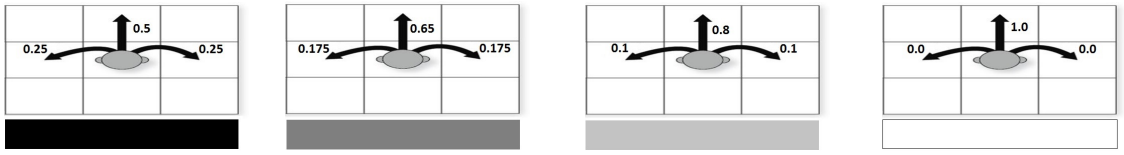


Figure 31: Light-Dark domain.



a) Transitions in black area.

b) Transitions in dark gray area.

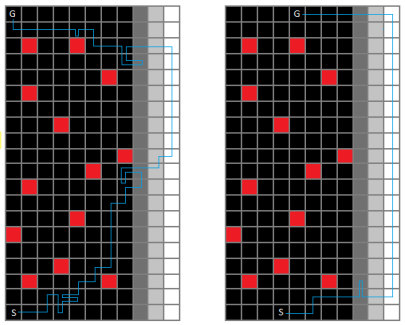
c) Transitions in light gray area.

d) Transitions in white area.

Figure 32: Stochasticity of action to move North in light-dark domain.

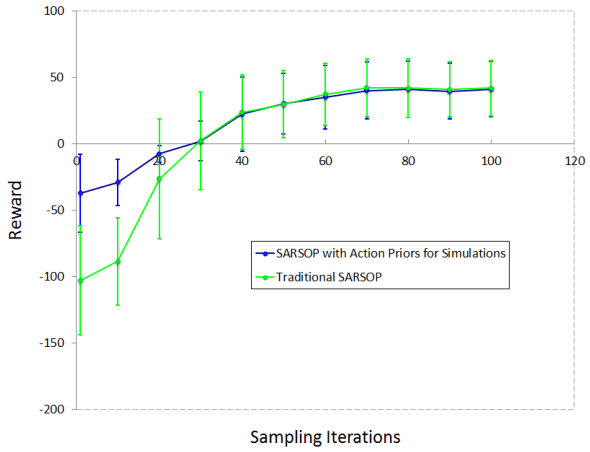
We note that the benefits afforded by the action priors are less than in the maze domain experiment, but still significant. This is due to the fact that the agent's actions

are highly stochastic in the darker regions of the domain, which means there is a high chance that despite taking the most useful action, the agent could possibly collide with a wall obstacle. However despite these dynamics we are able to observe an improvement as the action priors encourage movements towards regions where the agent experiences more light, to help solve tasks quicker as task execution is significantly easier when traveling in those regions (See Figure 33). This is evident as we can observe that the trajectories from the policies when using action priors are shorter and also allow an agent to complete a higher percentage of the tasks it is presented with (see Figure 34).

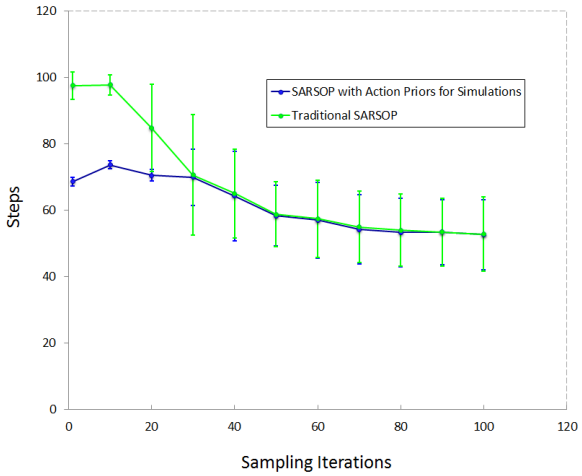


a) b)

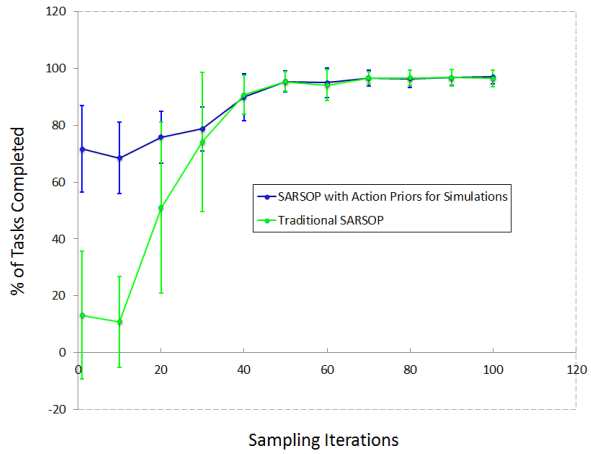
Figure 33: Examples of trajectories when using action priors.



a)



b)



c)

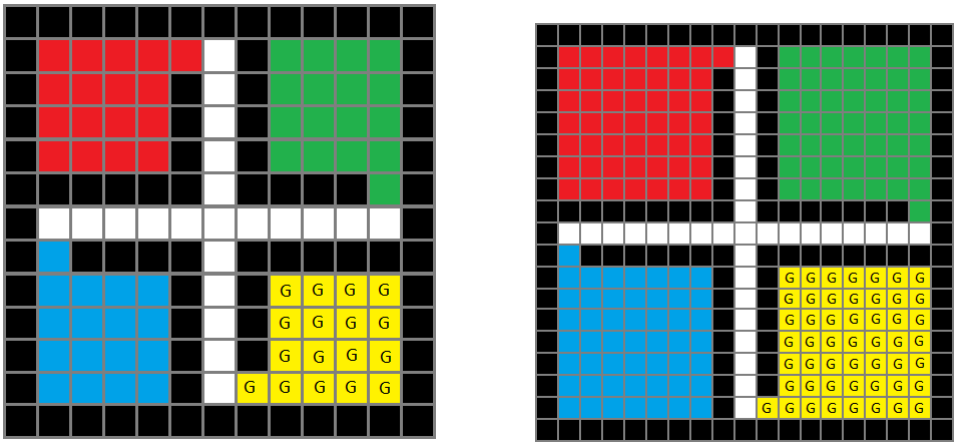
Figure 34: Performance results in light-dark domain. The results are averaged over 16 tasks.

4.2.4 Hallway Domain

We simulate a simple hallway problem where an agent is required to navigate from any area of the floor, other than that of the room to which the goal location resides, to the goal. Figure 35 shows a representation of the hallway domain which we used for the experiments that follow. The idea is to mimic a hallway which consists of 4 rooms where each enclosed colour symbolises a different room. The black grid cells represent walls, while every other colour represents passable space. The white grid cells represent the passage of the hallway, while the red, green, blue and yellow cells represent states belonging to a particular room. Each room has an entry or exit point through which the agent can travel to reach other areas of the domain. The grid cells marked with a "G" represent possible goal locations which means in all instances the agent will be required to travel to some randomly selected goal state in the room represented by the yellow grid cells. Figure 35a is the domain we used to learn action priors and Figure 35b is the domain we used to test whether or not the action priors will accelerate the learning of tasks. Note that the agent's perceptual contexts are consistent across the two domains.

With each episode of a task, the agent is initialised to some random location outside of the goal room. Each task will consist of the agent having to navigate from its initial position to some goal location in the goal room (room represented by the yellow colour in Figure 35) by taking a sequence of actions.

With regards to the agent's observation capability, we once again simulate a scaled down sonar by allowing the agent to be able to perceive only the eight grid cells that surround its current cell location (refer to Figure 27). However, the agent will additionally be able to precisely determine the room in which it is currently located or whether it is in the hallway space of the domain. Therefore, through the agent's observations, it will be able to determine the exact position of wall obstacles and whether or not it is in a particular room or the hallway. In other words, the observation space can be enumerated as 8 boolean flags, and one colour value.



a) Hallway domain used to learn action priors. b) Hallway domain used for testing action priors use.

Figure 35: The hallway domain.

As before, we first determine the action prior learning technique which produces the best results when incorporating action priors into SARSOP (see Figure 36). As in all prior domains, the POMDP method of learning action priors (Algorithm 8) outperforms the technique of learning action priors in an MDP (Algorithm 7). What is surprising at first glance is the extent to which the POMDP action priors outperform those learned from an MDP.

The combination of a good sensing capability and ability to gather a lot of action information from one task results in POMDP action priors that are far more superior than MDP action priors. By allowing the agent to be able to determine the exact location of walls and area of the maze in which it lies (i.e. the room in which it is located or whether it is in the hallway) these observation features allow us to establish informative action priors. Also, when we learn POMDP action priors we can run a large number of simulations to update the α_o-counts $\alpha_o^{\mathcal{P}}$ which allows us to obtain a lot of action information from just one task. Thus, a combination of using observation features that are capable of capturing a large amount of action information along with the ability to learn a lot of action information from just one task produces a very informative prior knowledge over actions.

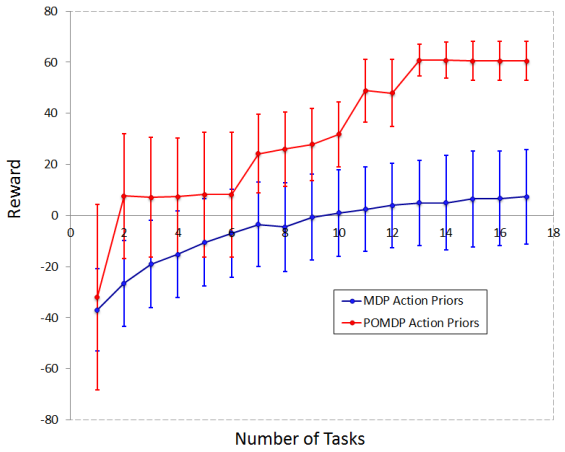
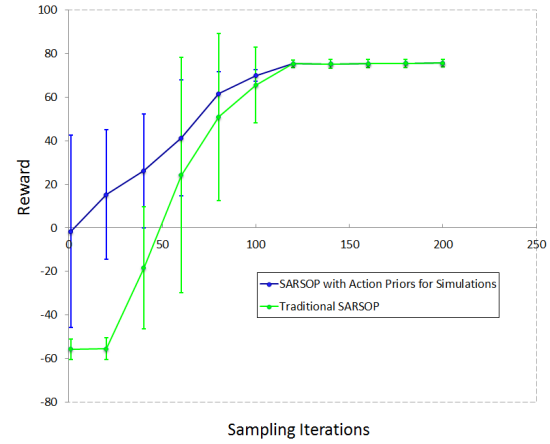


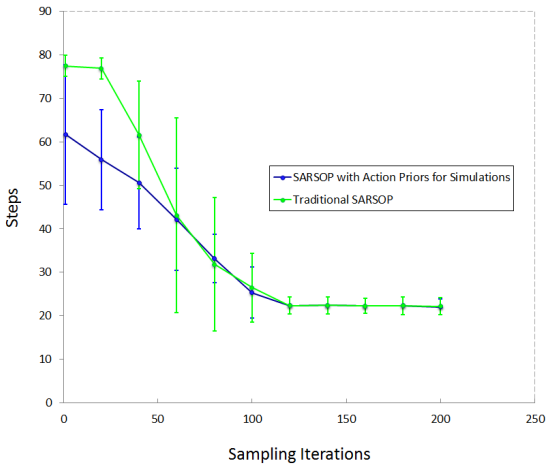
Figure 36: Action prior learning technique results in hallway domain. The results are averaged over 50 tasks.

Figure 37 shows the improvement in decision making at early phases of the learning process through the addition of action priors. One important observation is that SARSOP with Action Priors for Simulations immediately receives close to a positive return, unlike traditional SARSOP, as it has effectively learned to avoid harmful behaviours and only takes useful actions. Although in some instances, at early stages of learning, the agent may explore the domain and struggle to find the goal location within the goal room as suggested by Figure 37c, the action priors always enable the agent to avoid wall collisions and reach the goal room successfully.

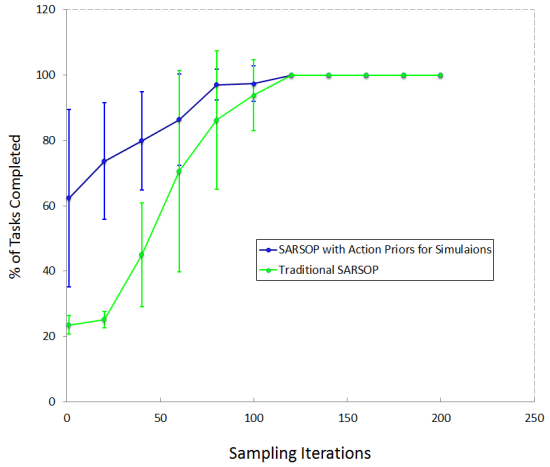
Another advantage that can be observed from these results is the ability of action priors to transfer knowledge to tasks with different state spaces and transition functions. This ability stems from the fact that the perceptual contexts of the agent are consistent across the two domains presented in Figure 35, even though the domains are different. This emphasises that the action prior knowledge learned for a set of observations is transferable to domains that are similar. This is possible because the action priors are defined over the observation space rather than the state space. Whether or not the observations are an absolute and complete description of the state of the environment, the action priors can still be successfully used to improve decision making. Similar conclusions can be made with regards to the experiments in Section 4.2.1 and 4.2.2 (see Figure 15 and 23).



a)



b)



c)

Figure 37: Performance results in hallway domain. The results are averaged over 50 tasks.

4.3 Conclusion

Planning with imperfect state information is a crucial capability for all autonomous robot agents operating in uncertain environments. However, considering imperfect state information during planning results in a much higher computational complexity when deciding which actions are best to take. This is because an agent can not choose the best action to take on the basis of a single known state, but rather depends on the set of all possible states consistent with the available information.

As a result of incorporating uncertainty into planning, the POMDP framework has led to improved performance in various robotic tasks. However, because of the computational challenges POMDPs face, a number of algorithms have surfaced in an attempt to improve the efficiency of using POMDPs. In particular, point-based POMDP solvers have made impressive progress in overcoming these challenges by computing approximately optimal solutions.

We consider the SARSOP algorithm, a point-based POMDP approach, as the basis of our research work. By identifying the complexity of solving POMDPs as a major issue, our goal is to illustrate the benefits of incorporating prior knowledge over actions in an uncertain environment. We consider uncertainty in robot control and sensor measurements to show the advantages of using action priors in this setting.

We develop three new algorithms called SARSOP with Action Priors for Sampling, SARSOP with Action Priors for Pruning and SARSOP with Action Priors for Simulations. However, to acquire the prior knowledge over actions to be used in these algorithms, an agent learns action priors using two different approaches. We allow an agent to learn action priors in an MDP and POMDP setting.

Through the use of experiments, we compare the performance of solving tasks using the traditional SARSOP with the SARSOP with action priors approaches. We also compare the quality of those solutions to assess which SARSOP approach gives better solutions. By using MDP action priors in one case and POMDP action priors in another, these comparisons also allow us to observe which action prior learning approach leads to better results.

After running various experiments, a number of conclusions can be made with regards to using action priors in a POMDP setting. Action priors indeed have the ability to improve the performance of learning tasks through the use of optimal behaviours from previously solved tasks (see Figure 19, 29, 30, 34 and 37). The action priors allow an agent to be equipped with a mechanism that allows the agent to accumulate general knowledge about a domain in the form of local behavioural invariances. The benefits of action priors symbolise an important form of knowledge transfer as they facilitate accelerated learning by biasing a decision making agent's search process towards previously useful choices. This was demonstrated through the use of the SARSOP with Action Priors for Simulations algorithm as it displayed the best results when incorporating action priors into SARSOP.

Throughout all the experiment domains used in in this chapter, the POMDP action prior learning technique of Algorithm 8 outperforms the MDP action prior learning

technique (Algorithm 7). This suggests that this is the better approach approach to use, between the two proposed methods, to learn priors when considering a POMDP problem. The POMDP method can also converge faster, using fewer learning tasks, than the MDP approach (see Figure 17 and 36 for an example of this). The MDP technique is disadvantaged as it requires more learning tasks to achieve the same results that the POMDP technique achieves using a smaller set of tasks.

We have also shown that the benefits of action priors can increase with an improved sensing capability (see Figure 28 and 29). By using an observation space that allows the agent to be able to precisely determine its surroundings, the action priors can be leveraged further. A more precise sensing capability allows an agent to gather more information about its current position in space which in turn improves the action priors' ability to classify the actions that are useful in certain situations.

The benefits of action priors rely heavily on the observational features it uses to define an observation when learning the priors. Therefore, the amount of action information that can be learned depends largely on the observation features it uses to define an observation.

There is a trade-off between the generality of the observations and the usefulness of the action priors. The more general the observation features, the less informative the action priors will be. However, the more specific the observational features are, the less portable the action priors are to new states. Note that the observational features the agent is able to use depend on properties of the task and domain.

Generally, action priors are particularly useful in rich environments that are characterised with repeating structure. Deciding which features should be used to define the set of observational features is an open question and depends on the capabilities of the agent. Feature learning in general has been considered in many contexts and is indeed an open and difficult question [Jong and Stone 2005][Lang and Toussaint 2009].

Perception-based action priors also have the ability to transfer knowledge to tasks with different state spaces and transition functions. This is possible because the perceptual contexts of the agent remain the same across trials, and therefore any action priors learned for these percepts can be transferred between trials.

The addition of action priors also have the ability to improve convergence speeds. This is achieved through the use of the SARSOP with Action Priors for Pruning algorithm. By formulating an additional pruning technique with respect to the action priors (see Algorithm 10, lines 10-12), we can observe a speed up in computational speeds. Using this action prior related pruning technique, the priors prune certain actions and provide a preference to actions based on the actions that were considered to be useful in previously solved tasks. This method of pruning allows the current observables to dictate useful actions.

Chapter 5

5 Related Work

Ultimately, the core theme of this thesis is abstraction, as appropriate abstractions enable knowledge and behaviours to be learnt in a generalised manner which in turn can be reused in new unexperienced scenarios. This research has focused on transferring knowledge learned from previously solved tasks to improve and accelerate the learning of subsequent tasks.

In this thesis, we use action priors to demostrate these ideas. However, there exists other research on learning policy priors and methods for transferring information from previous tasks to act as priors for a new task that have similar aspirations to our own. In this chapter we discuss some of these approaches and explain how they are similar to our work.

5.1 Action Transfer

Consider an MDP defined as a tuple (S, A, T, R, γ) (see Section 2.2.2). However, its use can be difficult for analysing knowledge transfer as it is not expressive enough to capture similarities across different task problems [Sherstov and Stone 2005]. Establishing a new MDP formalism that uses *outcomes* and *classes* can be used to overcome this difficulty. This is possible because the outcomes and classes can be used to remove the dependence of the model description (T and R) on the state set [Sherstov and Stone 2005].

Rather than specifying the effects of taking a particular action as a probability distribution $P(S)$ over next states, the effects of an action are specified as a probability distribution $P(\mathcal{O})$ over outcomes $\mathcal{O}$ [Boutilier et al. 2001]. Every action $a \in A$ has a probability distribution over $\mathcal{O}$. Therefore, the new transition function T is defined to be $T : S \times A \to P(\mathcal{O})$ where the range $P(\mathcal{O})$ is common to all tasks unlike $P(S)$. The effect of an outcome differs from state to state and the mapping of an outcome in a state to the actual next state is defined as part of the task description, rather than the domain definition.

Classes $\mathcal{C}$ are common to all task and specify an action's reward and transition dynamics in a state. Each state $s \in S$ in a task belongs to a class $c \in \mathcal{C}$. A group of states belong to the same class if an action's reward and transition dynamics are the same in those states i.e. $c(s_1) = c(s_2) \Rightarrow R(s_1, a) = R(s_2, a), T(s_1, a) = T(s_2, a)$ where $c(s_1)$ denotes the class of state s_1. The use of classes allows T and R to be defined over $\mathcal{C} \times A$, which is a set common to all tasks, rather than $S \times A$, which is a task-specific set. Thus, as a result of a combination of classes and outcomes the transition and reward function can be defined to be $T : \mathcal{C} \times A \to P(\mathcal{O})$ and $R : \mathcal{C} \times A \to \mathbb{R}$.

Using classes and outcomes, a domain can be defined to $D = (A, \mathcal{C}, \mathcal{O}, T, R)$ where A is a finite set of *actions* that the agent can take, $\mathcal{C}$ is a finite set of state *classes*, $\mathcal{O}$ is a finite set of action *outcomes*, $T : \mathcal{C} \times A \to P(\mathcal{O})$ is a *transition function* which gives the probability of an outcome when taking an action at a state belonging to a particular class and $R : \mathcal{C} \times A \to \mathbb{R}$ is a *reward function* that indicates the immediate payoff of taking an action at a state belonging to a particular class. A task M within domain D is defined by the tuple $M = (S, c, \eta)$ where S is a finite set of *states*, $c : S \to \mathcal{C}$ is *state classification function* and $\eta : S \times O \to S$ is a *next-state function.* This formalism allows us to distinguish related tasks to those that are not.

With regards to the related-task formalism, the maximum number of resulting next states from taking an action at a given state s is $|\mathcal{O}|$ states $(s_1, s_2, \ldots, s_{|\mathcal{O}|}$ where s_i represents the state that results if the ith outcome occurs). Considering the optimal values of these successor states, the resulting vector $\mathbf{v} = \langle V^*(s_1), V^*(s_2), \ldots, V^*(s_{|\mathcal{O}|}) \rangle$ is the *outcomes value vector* (OVV) of state s. The optimal action of an OVV of state s can be determined by

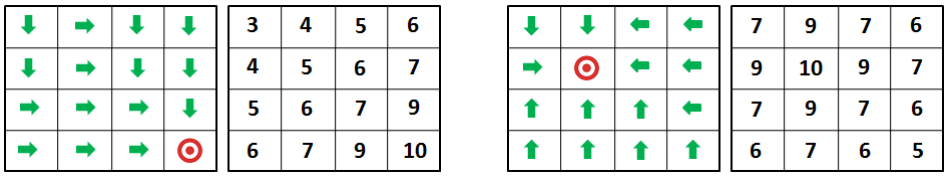
$$\pi^*(s) = \arg\max_{a \in A} \{R(c, a) + \gamma T(c, a) \cdot \mathbf{v}\}. \tag{53}$$

By grouping all OVVs of a task according to the class of each state, the set $U = \left\{U_{c_1}, U_{c_2}, \ldots, U_{c_{|c|}}\right\}$ denotes the set of OVVs of states from class c_i and determine the task's optimal action set.

Let $\widetilde{A}^* = \{a \in A : \pi^*(s) = a \text{ for some } s \in S\}$ be the *optimal action set* of an *auxiliary task*, and let $A^* = \{a \in A : \pi^*(s) = a \text{ for some } s \in S\}$ be the *optimal action set* of the *primary task*. In action transfer, $\widetilde{A}^*$ is learned in the auxiliary task which is then used to learn the primary task with the hope that $\widetilde{A}^*$ is simialr to A^*. Using transferred actions is feasible if every OVV in the primary task has in its vicinity an OVV of the same class in the auxiliary task [Sherstov and Stone 2005]. However, two dissimilar tasks can have very different OVV characteristics and thus different action sets.

Figure 38 shows an example of an auxiliary task that is dissimilar to the primary task. The target in Figure 38 represents the goal location, while the arrows represent the actions to be taken. Note that the optimal policy of the auxiliary task only includes South and East actions, while the optimal policy of the primary task includes North, East, South and West actions. Because the optimal policy of the primary task includes all four directions of travel, learning the primary task with actions transferred from the auxiliary task is therefore not feasible as the goal will be practically unreachable from most cells.

As such, through the use of transferring knowledge between related tasks, the learning process of MDP problems with large action sets can be accelerated [Sherstov and Stone 2005]. This is achieved by pruning the action set of an agent by using a technique that replaces its full action set with the optimal actions learned from an initial task. Subsequent tasks are learned using the reduced action set which reduces the complexity of the problem and facilitates faster learning. The overall idea of this technique is that if some action is relevant to some optimal policy of a particular task, then it is likely



Auxiliary task. Primary task.

Figure 38: An auxiliary and primary task, together with their optimal policies and value functions.

to be relevant in other tasks of that domain. This has very similar aspirations to the work in this thesis as it tries to cut down the action set or bias exploration in learning. However, the methods differ in that rather than using optimal policies to determine the relevance of an action or the reduced action set for each observation, the optimal policies are used as training data to determine the relevance of an action for the entire domain.

5.2 Shaping Rewards

Consider an MDP defined as a tuple (S, A, T, R, γ) (see Section 2.2.2). Intuitively, shaping refers to adding "shaping" rewards to the learning algorithm in an effort to guide it towards learning a feasible (or optimal) policy faster. Rather than running a reinforcement learning algorithm on (S, A, T, R, γ), it is run on some *transformed* MDP defined by the tuple (S, A, T, R', γ) where $R' = R + F$ is the reward function in the transformed MDP and $F = S \times A \times S \to \mathbb{R}$ is known as the *shaping reward function*. Therefore, instead of receiving a reward $R(s, a, s')$ for transitioning from state s to state s' after taking action a, the agent would receive $R(s, a, s') + F(s, a, s')$ for the same event in the transformed MDP.

$R' = R(s, a, s') + F(s, a, s')$ is the most general form of shaping rewards as it covers a fairly large range of possible shaping rewards one might come up with [Ng et al. 1999]. For example, consider a task where an agent is required to travel to a goal from its starting position by taking a sequence of actions. A shaping reward function that may be useful for this task is

$$F(s, a, s') = \begin{cases} r & \text{whenever } s' \text{ is closer to the goal than } s \\ 0 & \text{otherwise} \end{cases} \tag{54}$$

where r is some positive reward. Another example of a shaping reward function would perhaps be to encourage the selecting of an action a_1 in a particular state s. To accomplish this, one might set

$$F(s, a, s') = \begin{cases} r & \text{whenever } a = a_1 \text{ in state } s \\ 0 & \text{otherwise} \end{cases} \tag{55}$$

Note that in many reinforcement learning applications, an agent is allowed to learn the system dynamics through experience (i.e. through taking actions in the MDP and observing the resulting state transitions and rewards). In this case, the agent is not explicitly given the MDP as a tuple (S, A, T, R, γ). Given such access to the MDP, the transformed MDP can be simulated to have the same type of access by simply taking actions and acting as if the agent has observed reward $R(s, a, s') + F(s, a, s')$ whenever the reward $R(s, a, s')$ is actually observed in the original MDP. This is possible because the original MDP and the transformed MDP use the same states, actions and transition probabilities. Thus, the online/offline model-based/model-free algorithms that may be applied to the original MDP may in general be readily applied to the transformed MDP in a similar fashion.

With regards to accelerating reinforcement learning in general and particularly in goal-directed exploration, shaping can be found to be a popular method [Dorigo and Colombetti 1998]. Shaping allows the agent's reward function to be augmented through the use of intermediate shaping rewards or "progress indicators" that provide a more informative reinforcement signal to the agent [Matarić 1997]. However, an externally specified shaping reward function to be included in a reinforcement learning system can require significant engineering effort [Sutton and Barto 1998]. Thus, an easier approach may be to allow an agent to learn its own shaping function from experience across several tasks without having to have it specified in advance.

By learning which sensory patterns predict reward across tasks, this information can be used as a shaping function that provides a first estimate for the value of newly discovered states when learning a value function for an unexperienced task. In this case, an agent would initially have some precified, possibly random or uniform, shaping function and then refine it as it experiences several related but distinct task instances over its lifetime. Distinct related task instances refer to variations of the same type of task which have some commonality between them so that as the agent solves each task it is able to retain learned knowledge usefully across them.

Reward linked related tasks are defined as follows. When considering a variation of tasks, the agent creates two representations namely a *problem-space* and an *agent-space*. The problem-space refers to the state-space along with its accompanying transition probabilities and reward function which is different for each task. The agent-space refers to the sensations that are consistently present and retain the same semantics across all variations of a task. Thus, the variations of a task are called *related* if the tasks consist of environments that share an agent-space. The task variations are called *reward-linked* if the reward function in each environment allocates the same rewards for the same type of events (e.g. regardless of which environment the agent is in, the reward is always r_1 for bumping into a wall and r_2 for reaching a light source). The purpose this serves is to ensure that there is some useful relationship and potential correlation between the sensations in the agent-space and reward across task variations, which the agent can learn to exploit to accelerate learning in other tasks.

Ultimately, shaping rewards present another use of knowledge transfer to accelerate learning in subsequent related tasks. Reward shaping includes learning a reward

predictor learned from using prior experience on a sequence of tasks [Konidaris and Barto 2006]. The idea is to learn a measure that provides more information over the reward that will accelerate the learning of more difficult tasks. This is accomplished by maintaining a problem-space and an agent-space. The problem space refers to a Markov representation that differs for each task, while the action space is where value predictions are formed. Similarly, learning action priors is related to the idea of maintaining a problem space and an agent space. In this case, the agent space refers to the common elements between tasks, while the problem space is specific to each task.

5.3 Options

An *option* is a learned abstract action in a semi-MDP model [Precup et al. 1998]. Options are used in semi-MDP models because they violate the Markov assumption (see Section 2.2.1). This violation stems from the fact that the options take arbitrarily long to execute preventing the probability of the next time step's state to depend only on the action and the current state.

An option is an already learned policy for performing a subtask that could be transferred and used in solving other tasks as an additional action in the MDP model. For example, consider a robot that has learned to drive a car from one point to another. Thus, it could use that as one of its skills or actions when learning to plan for another task such as having to deliver goods from the factory to a store.

Because each option is a regular policy that utilises lower-level actions to execute, the learning of the optimal policy for MDPs that implement options is therefore very similar to the original case. After each option is learned, some statistics such as the expected reward and transition probabilities are computed and preserved. The expected reward function $R(s, o)$ is defined to be the sum of the discounted rewards that could be obtained by executing option o in state s, while the transition function $T(s, o, s')$ gives the probability of option o terminating in state s' if o is executed in state s. The reward and transition functions for options behave similarly to those of regular actions and can be incorporated into the model's set of actions. They can be implemented into the learning of subsequent tasks by simply making slight modifications to the algorithms and the model [Precup et al. 1998][Janzadeh and Huber 2013].

Similar to regular POMDP policies (see Section 2.3.3), POMDP options can be defined by a finite set of linear functions represented by vectors. These are known as β-vectors and denote the immediate reward for following a given option at a particular belief state.

However, an option requires a method which will allow it to terminate as its execution time can be infinite after being selected. One way to achieve this is by incorporating a terminating action to the option's action set that can act as a deterministic termination function. Careful consideration needs to be taken into account to achieve this as the reward function has to be manipulated so as to ensure that the termination action is selected when the subtask is completed. A similar process for initialisation has to be followed as initialisation has to ensure that an option is not selected in a belief state

outside its *initiation set*. An initiation set is a set consisting of all states in which the option can be initiated.

Although there has been a great deal of research in the use of abstraction and its benefit with regards to transferring knowledge learned from previous tasks to new tasks, very little has been studied in a POMDP context. However, options can be successfully used to transfer skills learned from previous experiences to improve the learning speeds of other POMDPs [Janzadeh and Huber 2013]. Options are high level actions and involve learning a policy for executing a subtask that can be transferred and used as an added action in the POMDP model to solve other tasks. Learning actions priors can be seen to share similar characteristics to learning the initiation sets of options. However, they can be differentiated in that action priors describe areas where the option is sensible/useful, while the initiation sets of options define where the option can physically be instantiated/used.

5.4 Policy-Contingent Abstraction (PolCA+)

By leveraging domain knowledge and setting intermediate goals to solve a complex task, *hierarchica*l MDP approaches accelerate the planning of complex planning tasks. There exists a number of hierarchical MDP approaches namely, Hierarchical Abstract Machine (HAM) [Parr and Russell 1998], ALisp [Andre and Russell 2002] and MAXQ [Dietterich 2000]. These approaches include defining seperate subtasks, which are handcrafted, and constrain the solution search space. Subtasks are defined by a set of initial states, a set of goal states, reduced action sets, fixed/partial policies and local reward functions (i.e. reward functions defined for each subtask).

Generally, in hierarchical MDP problem solving, it is not necessary for each subtask planner to consider every state. This is because a collection of states can be grouped into one by appropriately igonring irrelevant features. By doing so, subtask optimisation can be accelerated without affecting the quality of the policy. For example, consider a robot that has to travel from one point to another, but learns which direction to travel in order to complete the task by conversing with pedestrians. In this case, it is not necessary to consider the robot's precise (x, y) location coordinates when selecting what to say to a pedestrian.

Applying subtask-specific abstraction is well known [Dietterich 2000][Andre and Russell 2002], and most hierarchical approaches require a handcrafted state abstraction for each subtask. However, designing subtask-specific abstraction requires significant engineering effort and can be difficult to correctly formulate. Additionaly, it is also unable to leverage policy-specific abstraction opportunies. For example, to which extent can the robot's (x, y) location coordinates be abstracted for conversation tasks as the robot's distance from a pedestrian can make conversing very difficult. Therefore, the exact amount or form of abstraction can be hard to quantify.

Fortunately, there exists research that focuses on automatically learning *state abstraction functions* for non-hierarchical MDP problems [Dean and Givan 1997], which can be adopted by hierarcical MDPs. Automatically discovering a state abstraction

function includes learning a function that maps states to *clusters of states* so that an agent can learn a policy over clusters of states. However, the challenge lies in discovering a grouping that significantly reduces planning time and allows planning over clusters with minimal loss of performance when compared to planning over the entire state space.

Policy Contingent Abstraction (PolCA) relies on a set of basic structural assumptions and builds on concepts found in both hierarchical MDP algorithms and automated state abstraction techniques. These structural assumptions are similar to other hierarchical MDP approaches and are composed of a task graph. Formally, a task graph G includes leaf nodes, where each leaf node represents a *primitive* action a from the original action set A, and internal nodes, where each internal node has the role of representing both a distinct subtask g as well as an *abstract* action $\widehat{a}$. Each distinct subtask g has an action set defined by its children in the hierarchy, while the abstraction action $\widehat{a}$ is defined in the context of the above-level subtask.

Formally, subtask g is defined by $A_g = \{\widehat{a}_j, \ldots, a_p, \ldots\}$ the set of actions which are allowed in subtask g and $R_g(s, a)$ the local reward function [Pineau et al. 2003]. As a result of the hierarchy, there is one action for each immediate child of g. Each subtask in the hierarchy must have local (non-uniform) reward in order to optimize a local policy, which in general is equal to the true reward $R(s, a)$. However, an addition of a pseudo-reward that specifies the desirability of achieving the subgoal is required in subtasks where all available actions have equal reward (over all states).

Note that we also require a model of the domain $D = (S, A, T, R, \gamma)$. Given an MDP (S, A, T, R, γ) and a task hierarchy $G = \{g_0, \ldots, g_n\}$, planning with PolCA requires four steps to learn a policy. In order, these are: structure the state space [Parr and Russell 1998]; for each subtask $g \in G$ in bottom-up order, parameterise the subtask; cluster the subtask; and solve the subtask [Pineau et al. 2003].

By leveraging the structural assumptions of PolCA and extending it to POMDPs, the result is a scalable hierarchical POMDP algorithm called PolCA+. PolCA+ can not guarantee recursive optimality as a result of the properties of belief space planning [Pineau et al. 2003]. However, because of its ability to handle partial observability, it is therefore much better suited for real-world robotic tasks over its MDP counterpart PolCA.

The Policy-Contingent Abstraction algorithm (PolCA+) is another approach to reusing experience in an effort to accelerate the learning of policies in a POMDP context [Pineau et al. 2003]. It achieves this by learning task-specific abstractions. This involves using a task hierarchy to learn an abstraction function and a recursively-optimal policy for each subtask as it traverses from the bottom to the top of the hierarchy. Through its use of task partitioning and task-specific state abstraction, PolCA+ is able to find good policies for large POMDP problems in a shorter period of time. On the contrary, action priors focus on learning domain invariances to facilitate faster learning through forming better abstractions of the domain, rather than learning task-specific abstraction and policies to accelerate the learning of policies.

5.5 HQ-Learning

HQ-learning makes use of the notion that only a few memories corresponding to important previously achieved subgoals can be sufficient to select the next optimal action. For example, consider an agent that was given instructions to travel from one point to another. If its instructions were "head north until a beacon is reached, then turn east and travel in that direction until another beacon is reached and then travel in a northerly direction until the target destination is reached." During the agent's travel from its start position to the destination, only a few memories are significant or relevant to succesfully complete the task. An example of such a relevant memory is "the first beacon has already been passed". This memory gives sufficient information to the agent that it should continue travelling in an easterly direction until another beacon is reached. Therefore, between two subgoals a memory-independent *reactive policy* will successfully guide the agent to completing its task.

HQ-learning decomposes a given POMDP into a sequence of *reactive policy problems* using a divide and conquer strategy. Reactive policy problems are solved by reactive policies. Any deterministic finite POMDP optimal POMDP policy with a fixed starting state and goal state can be decomposed into a finite sequence of optimal reactive memoryless policies. Each subgoal determines the transition from one reactive policy to another. Reactive policies allow all states causing identical inputs to require the same optimal action. Thus, the only points that are of importance are those that correspond to transitions from one reactive policy to the next.

HQ-learning uses a number of reactive policy problem solving subagents to handle the transitions from one reactive policy to another. Only one subagent can be active at a given time and each subagent's reactive policy consists of an adaptive mapping from observations to actions. HQ-learning's only type of memory is embodied by a pointer that indicates which subagent is currently active. This means, the internal system state of HQ-learning is just a pointer to which subagent is active.

The reactive policies of different subagents are comdined in a manner in which is learned by the subagents themselves. Initially, the first active subagent generates a subgoal for itself using its *HQ-table*, which is a subgoal table [Wiering 1999]. Thereafter, it follows the policy embodied by its Q-function until it successfully completes its subtask. As soon as it achieves its subgoal, control is passed to the next subagent and the procedure repeats itself until the overall goal is achieved or a time limit is reached. When this procedure terminates, each agent adjusts both its reactive policy and its subgoal using two learning rules that interact without explicit communication. The first states that Q-table adaptation is based on slight modifications of Q-learning and the second states that HQ-table adaptation is based on tracing successful subgoal sequences by Q-learning on the higher (subgoal) level [Wiering 1999]. This allows subgoal (or reactive policy) combinations that lead to higher rewards to become more likely to be chosen.

Formally, there is an ordered sequence of n subagents $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$, where $\mathcal{P}_1$ corresponds to the subagent that will solve subtask 1, and $\mathcal{P}_2$ will solve subtask 2

and so fourth. Each subagent $\mathcal{P}_i$ is equipped with a Q-table, an HQ-table and a control transfer unit that transfers control from one subagent to another, with the exception of $\mathcal{P}_n$ which only has a Q-table. A subagent's Q-table (given by a matrix of size $|A| \times |O|$, where $|A|$ is the number of possible actions and $|O|$ is the number of different observations) represents its local policy for executing an action given an input. $Q_i(o_t, a_j)$ denotes $\mathcal{P}_i$'s utility (Q-value) of selecting action a_j when observation o_t is observed. Here, t represents time steps $t = 1, 2, \ldots, k$. Its HQ-table is a vector with $|O|$ elements which generates the subagent's current subgoal. This means, each HQ-table entry represents the estimated value of a possible observation being a subgoal. Therefore, $HQ_i(o_j)$ denotes $\mathcal{P}_i$'s HQ-value (utility) of selecting o_j as its subgoal. Using a function $Active(t) = i$, the system is able to determine if subagent $\mathcal{P}_n$ is active at time step t which, as stated before, is HQ-learning's only type of memory.

When selecting a subgoal at the beginning, $\mathcal{P}_1$ is made active. When $\mathcal{P}_i$ is made active, its HQ-table is utilised to choose a subgoal for $\mathcal{P}_i$. A Max-random rule is used to explore different subgoal sequences and states that the subgoal with maximal HQ_i value is selected with probability P_{max} and a random subgoal is selected with probability $1 - P_{max}$ [Wiering 1999]. In situations where there are multiple subgoals with maximal HQ_i-values, subgoals are randomly selected among that group of subgoals.

When selecting an action, the subagent's action choice depends purely on the current observation o_t. The Max-Boltzmann distribution is used to select actions during the learning process [Wiering 1999]. Primarily, this is used to adjust the degree of randomness involved in the subagent's action selection procedure.

In order to transfer control from one active subagent to another, the following process is used. After each action execution by a subagent, the subagent uses its control transfer unit to assess whether $\mathcal{P}_i$ has reached the goal. If it hasn't, its control transfer unit assesses whether $\mathcal{P}_i$ has achieved its subgoal to decide whether control should be passed on to $\mathcal{P}_{i+1}$.

HQ-learning can be used for inductive transfer. Inductive knowledge transfer in POMDPs may also facilitate generalisation and knowledge transfer to accelerate the learning of subsequent tasks, as long as different world states leading to similar outcomes require similar treatment [Wiering 1999].

Incremental learning can be used to transfer the experience from a relatively simpler domain's solution to a more complex one that has similar characteristics. In this case, an agent is trained to solve a task in a simpler domain and then without resetting its Q-tables and HQ-tables, the agent is then required to solve a task in a more complex domain. Q-tables are used to select the next action and are stored estimates of actual observation/action values, while HQ-tables are used to generate a subgoal once the agent is made active and are stored estimated subgoal values. This is similar to our work as we learn action priors in a smaller, but similar domain and transfer the knowledge learned from solving the tasks in that domain to new tasks in a more complex domain that has similar charateristics to accelerate the learning of these new tasks. However, using action priors differs in that the priors are learned from multiple tasks as opposed to solving one task and transferring that knowledge to a new task.

Chapter 6

6 Future Work

The same ideas demonstrated here with SARSOP can be applied to other point-based POMDP approaches, where the incorporation of action priors would be able to prune actions or prioritise action selection in different learning instances. However, the domains used with these approaches must support multiple tasks, with related structure.

It would be particularly interesting to use these principles of action prior use in online POMDP paradigms, such as the Determinized Sparse Partially Observable Tree (DESPOT) algorithm [Somani et al. 2013], with a larger action space. We believe an agent stands to make significant gains in this context as reasoning over a wide branching factor of different action sequences can be very computationally demanding. However, the action priors should be able to guide the search process towards previously experienced solutions with less computational effort.

In most cases, existing point-based POMDP approaches assume a discrete state space, while in reality the natural state space for a robot agent is often continuous. The curse of dimensionality is a common difficulty when the state space is continuous as there are infinitely many states in a continuous state space and you still need to keep a continuous belief over them. However, a powerful technique for handling this issue is through the use of probabilistic sampling [Traub and Werschulz 1998]. Using action priors to guide this sampling process would possibly have the advantage of improving this method by basing its sampling on behaviours that were considered to be optimal in previously solved tasks.

Another possible avenue that could be worth investigating in POMDPs and could possibly demonstrate further advantages is by learning action priors over the belief space. In our work we learned action priors based on observations, but an alternative for POMDPs would be to base the priors on belief states. Such a basis might also yield some interesting insights given the continuous and structured nature of belief states. An investigation into whether there were advantages to either direction (observation-based or belief-based) and under which situations would be useful.

In the same way as we can define preferences over actions based on states or observations, we could possibly do this over beliefs. In this case, we would possibly need to work with some approximation (possibly leveraging the SARSOP framework). What makes this more interesting is that the action priors are then defined over a continuous space. Using the ideas of SARSOP, perhaps there could be pruned regions or vectors in the belief space.

We developed three algorithms, namely SARSOP with Action Priors for Sampling, SARSOP with Action Priors for Pruning and SARSOP with Action Priors for Simulations (see Section 3.3.2). Perhaps considering a combination of these approaches may

yield benefits. For example, developing a new algorithm that combines SARSOP with Action Priors for Sampling with SARSOP with Action Priors for Pruning. In this case, we would essentially use action priors for selecting actions during the sampling method of SARSOP as well as to prune α-vectors on top of SARSOP's δ-dominance pruning technique.

Action priors build a model of "common sense" behaviour of a domain through the combination of domain invariances and the common elements of policies used to solve a variety of different tasks in the same domain. In this thesis, we only used this approach to address the idea of accelerating the learning of new tasks in the same agent. However, an interesting idea to address is to demonstrate how the same methodology can be used in a multi-agent setting. In this case, the action prior knowledge learned from one agent is transferred to another, based on the knowledge it has learned from the behaviour of numerous other agents.

Another interesting study that could be carried out is the use of function approximation to formulate the MDP action priors. Linear function approximation involves establishing the value function as a linear combination of a set of basis functions, which are refered to as features [Gordon 1995][Busoniu et al. 2010]. The suspicion is that the feature choices in these maze environments will have an impact on performance.

Perhaps a comparative analysis with the existing literature presented in Chapter 5 could help in consolidating the performance of our model. This would entail running experiments using the various approaches on the same domain environments and comparing the effectiveness of these algorithms in improving the learning process using knowledge transfer. This would allow us to determine the most fruitful approach for using knowledge transfer to accelerate the learning of future tasks.

In this thesis we are not concerned with the time in which it takes to learn the action priors as we consider it to be a once-off overhead. However, the exclusion of this work leaves the reader without information on how much prior work would have to be completed to appreciate the faster convergence time. A key point to consider here would be to emphasise the tradeoff between the effort used to learn the action priors versus the appreciation for the faster convergence time. Less computational effort would reflect a greater appreciation for the faster convergence time, while an increased computational effort may result in a decreased appreciation.

Lastly, these ideas could be carried out on a real robot to demonstrate that these ideas work effectively not only in theory but practically as well. This would include carrying out simulations using simulation environments, such as ROS, and then carrying that work over to a real life implementation using a real robot and domain. Additionally, perhaps simulations could be used to learn action priors which, in turn, would be transferred to physical domains where they are used on a real robot.

Chapter 7

7 Conclusion

This research was largely concerned with learning general domain knowledge, particularly in the context of partial observability, and transferring this knowledge to foreign tasks to be solved in future. The purpose of transferring this knowledge is to accelerate the completion of unfamiliar tasks by reusing both this learned knowledge and previously learned behaviours. The goal is to enable an agent to be capable of making competent decisions even in the early phases of a learning process.

We propose that for an agent to be generally capable when it is required to perform a range of unknown tasks in which it experiences uncertainty, the agent must have the ability to continually learn from a lifetime of experience which is largely dependent on its ability to generalise from past experiences and its ability to form representations which facilitate faster learning and the transfer of knowledge between different situations. By exploiting the commonalities between large families of tasks, the agent may potentially minimise situations where it has to relearn from scratch. This also has the effect of allowing an agent to build better models of the domain.

We propose a method that extends existing POMDP formalisms by taking into account the statistics of action choices over a lifetime of the agent, in an effort to allow the agent to be able to quickly cut down options when deciding which actions to select, in a manner which may not have been obvious if each task was solved in isolation. We show how the agent can reuse behavioural knowledge learned from previously solved tasks to accelerate the learning of new tasks.

We have demonstrated the advantages of abstraction and its strengths in allowing an agent to learn and make feasible decisions quickly in unforeseen situations. To a large extent, this thesis focused on how a learning agent can acquire a form of "common sense" knowledge learned from multiple instances of different tasks in the same domain. This knowledge provides a set of behavioural guidelines allowing an agent to be able to appropriately handle both familiar and unfamiliar tasks. By allowing an agent to perform multiple tasks in an environment, it can learn domain invariances by forming better abstractions of the environment as a result of considering the underlying behavioural commonalities among these experienced tasks.

Choosing the best action to take in situations where agents have a wide variety of actions can be a difficult task. However, by learning the task-independent behavioural commonalities of an environment, an agent can focus its exploratory behaviour away from actions that are detrimental or useless. By transferring the knowledge learned from previously solved tasks to solve new tasks, an agent can be able to determine the behaviours that are more suitable in particular situations. This is essentially the idea behind the use of action priors.

Action prior knowledge has the ability to improve the performance of learning tasks as they are able to specify action usefulness and guide exploration towards behaviours that have been useful in previously solved tasks. In this thesis, we have demonstrated how these priors can be advantageous in a POMDP context and how their use can lead to the computation of good policies in a shorter period of time.

The knowledge given by the action priors is based on previous experiences of the agent in an environment and can be learned by taking into account the behaviours that were considered to be optimal when learning previously experienced tasks. They are learned from solving multiple tasks, and by studying the optimal policies that arise from solving those tasks, we are able to learn about the underlying structure of the domain the agent is acting in.

By learning the invariances of an environment, an agent is able to gain insights into learning what not to do in particular situations. This has the ability of guiding the exploration process away from risky or unsafe behaviours as the priors bias search towards previously useful choices.

Not only do action priors allow an agent to be able to transfer knowledge between different tasks in the same domain, but enables cross domain transfer as the priors may also be used in different state spaces. This is because the priors are associated with observations, allowing an agent to determine the useful actions based on its current observables. However, this is particularly fruitful if the domains have the same observation space.

This thesis presents a step towards autonomous robots that can operate reliably in uncertain environments. We hope that future work can build on the ideas presented in this thesis because an essential capability for robots as such is their ability to handle uncertainty for efficient task planning with imperfect state information.

References

[Andre and Russell 2002] D. Andre and S. Russell. State Abstraction for Programmable Reinforcement Learning Agents. *Proceedings of the AAAI*, 2002.

[Bishop 2006] C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Bonet 2009] B. Bonet. Deterministic POMDPs Revisited. *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*, pages 59-66, 2009.

[Busoniu et al. 2010] L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst. Reinforcement Learning and Dynamic Programming Using Function Approximators, 39, CRC press, 2010.

[Boutilier et al. 2001] C. Boutilier, R. Reiter, and B. Price. Symbolic Dynamic Programming for First-Order MDPs. *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 690-697, 2001.

[Braziunas 2003] D. Braziunas. *POMDP Solution Methods*. Technical Report, Department of Computer Science, University of Toronto, 2003.

[Cassandra et al. 1996] A.R. Cassandra, L.P. Kaelbling, and J.A. Kurien. Acting under Uncertainty: Discrete Bayesian Models for Mobile-Robot Navigation. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2, pages 963-972, 1996.

[Cassandra et al. 1997] A.R. Cassandra, M.L. Littman, and N.L. Zhang. Incremental Pruning: A Simple, Fast, Exact Method for Partially Observable Markov Decision Processes. *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 54-61, 1997.

[Cassandra 1998] A.R. Cassandra. *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*. Ph.D. Thesis, Brown University, 1998.

[Cassandra et al. 1994] A.R. Cassandra, L.P. Kaelbling, and M.L. Littman. Acting Optimally in Partially Observable Stochastic Domains. *Proceedings of the AAAI*, 94, pages 1023-1028, 1994.

[Dean and Givan 1997] T. Dean and R. Givan. Model Minimization in Markov Decision Processes. *Proceedings of the AAAI*, 1997.

[Dietterich 2000] T.G. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 13, pages 227-303, 2000.

[Dorigo and Colombetti 1998] M. Dorigo and M. Colombetti. *Robot shaping: An Experiment in Behavior Engineering*. MIT Press/Bradford Books, 1998.

[Galleguillos and Belongie 2010] C. Galleguillos and S. Belongie. Context Based Object Categorization: A Critical Survey. *Computer Vision and Image Understanding*, 114.6, pages 712–722, 2010.

[Gordon 1995] G.J. Gordon. Stable Function Approximation in Dynamic Programming. *Proceedings of the 12th International Conference on Machine Learning*, 1995.

[Guestrin et al. 2001] C. Guestrin, D. Koller, and R. Parr. Solving Factored POMDPs with Linear Value Functions. *Seventeenth International Joint Conference on Artificial Intelligence workshop on Planning under Uncertainty and Incomplete Information*, pages 67-75, 2001.

[Guy et al. 2013] S. Guy, J. Pineau, and R. Kaplow. A Survey of Point-Based POMDP Solvers. *Autonomous Agents and Multi-Agent Systems*, 27.1, pages 1-51, 2013.

[Hauskrecht 2000] M. Hauskrecht. Value-function Approximations for Partially Observable Markov Decision Processes. *Journal of Artificial Intelligence Research*, 13, pages 33–94, 2000.

[Hsiao et al. 2007] K. Hsiao, L.P. Kaelbling, and T. Lozano-Perez. Grasping POMDPs. *Proceedings of the IEEE International Conference on Robotics & Automation*, pages 4685–4692, 2007.

[Hsu et al. 2008] D. Hsu, W.S. Lee, and N. Rong. A Point-Based POMDP Planner for Target Tracking. *Proceedings of the IEEE International Conference on Robotics & Automation*, pages 2644–2650, 2008.

[Hu et al. 2003] X. Hu, D.F. Alarcon, and T. Gustavi. Sensor-Based Navigation Coordination for Mobile Robots.*Proceedings of the 42nd IEEE Conferenee on Decision and Control*, pages 6375–6380, 2003.

[Janzadeh and Huber 2013] H. Janzadeh and M. Huber. Learning Policies in Partially Observable MDP with Abstract Actions Using Value Iteration. *FLAIRS Conference*, 2013.

[Jong and Stone 2005] N.K. Jong and P. Stone. State Abstraction Discovery from Irrelevant State Variables. *International Joint Conference on Artificial Intelligence*, pages 752–757, 2005.

[Kaelbling et al. 1998] L.P. Kaelbling, M.L. Littman, and A.R. Cassandra. Planning and Acting in Partially Observable Stochastic Domains. *Artificial intelligence*, 101.1, pages 99-134, 1998.

[Kelly et al. 2006] A. Kelly, A. Stentz, O. Amidi, M. Bode, D. Bradley, A. Diaz-Calderon, M. Happold, H. Herman, R. Mandelaum, T. Pilarski, P. Rander, S.

Thayer, and R. Warner. Toward Reliable off Road Autonomous Vehicles Operating in Challenging Environments. *Journal of Field Robotics*, 25, pages 449–483, 2006.

[Klein and Abbeel 2013] D. Klein and P. Abbeel. *CS188: Artificial Intelligence.* Lecture Slides, Department of Computer Science, U.C. Berkeley, 2013.

[Koenig and Simmons 1998] S. Koenig and R.G. Simmons. Xavier: A Robot Navigation Architecture Based on Partially Observable Markov Decision Process Models. *Artificial Intelligence Based Mobile Robotics: Case Studies of Successful Robot Systems,* pages 91–122, 1998.

[Konidaris and Barto 2006] G.D. Konidaris and A.G. Barto. Autonomous Shaping: Knowledge Transfer in Reinforcement Learning. *Proceedings of the 23rd International Conference on Machine Learning*, pages 489–496, 2006.

[Kurniawati et al. 2008] H. Kurniawati, D. Hsu, and W.S. Lee. SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces. *Proceedings of Robotics: Science and Systems*, 2008.

[Kurniawati et al. 2010] H. Kurniawati, Y. Du, D. Hsu, and W.S. Lee. Motion Planning under Uncertainty for Robotic Tasks with Long Time Horizons. *The International Journal of Robotics Research*, 2010.

[Lankenau and Meyer 1999] A. Lankenau and O. Meyer. Formal Methods in Robotics: Fault Tree Based Verification. *Proceedings of Quality Week*, 1999.

[Lim et al. 2011] Z.W. Lim, D. Hsu, and W.S. Lee. Monte-Carlo Planning in Large POMDPs. *Proceedings of Advances in Neural Information Processing Systems (NIPS-2011),* 2011.

[Littman et al. 1995] M.L. Littman, A.R. Cassandra, and L.P. Kaelbling. Learning Policies for Partially Observable Environments: Scaling Up. *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362-370, 1995.

[Littman 1996] M.L. Littman. *Algorithms for Sequential Decision Making.* Ph.D. Thesis, Department of Computer Science, Brown University, 1996.

[Maaref and Barret 2002] H. Maaref and C. Barret. Sensor-Based Navigation of a Mobile Robot in an Indoor Environment. *Robotics and Autonomous Systems*, 38, pages 1-18, 2002.

[Matarić 1997] M. Matarić. Reinforcement Learning in the Multi-Robot Domain. *Autonomous Robots*, 4, pages 73-83, 1997.

[Mitchell 1997] T.M. Mitchell. *Machine Learning.* Burr Ridge, IL: McGraw Hill, 45, 1997.

[Morris 2007] A.C. Morris. *Robotic Introspection for Exploration and Mapping of Subterranean Environments*. Ph.D. Thesis, Carnegie Mellon University, 2007.

[Murphy 2000] K.P. Murphy. *A Survey of POMDP Solution Techniques*. Technical Report, Department of Computer Science, U.C. Berkeley, 2000.

[Ng et al. 1999] A. Ng, D. Harada, and S. Russell. Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. *Proceedings of the 16th International Conference on Machine Learning*, pages 278-287, 1999.

[Ong et al. 2009] S. Ong, S. Png, D. Hsu, and W. Lee. POMDPs for Robotic Tasks with Mixed Observability. *Proceedings of Robotics: Science and Systems*, 29.8, pages 1053-1068, 2009.

[Papadimitriou and Tsitsiklis 1987] C.H. Papadimitriou and J.N. Tsitsiklis. The Complexity of Markov Decision Processes. *Mathematics of Operations Research*, 12.3, pages 441–450, 1987.

[Parr and Russell 1998] R. Parr and S. Russell. Reinforcement Learning with Hierarchies of Machines. *Advances in Neural Information Processing Systems*, 10, pages 1043-1049, 1998.

[Parr and Russell 1995] R. Parr and S. Russell. Approximating Optimal Policies for Partially Observable Stochastic Domains. *Proceedings of the International Joint Conference on Articial Intelligence*, pages 1088-1094, 1995.

[Pineau et al. 2003] J. Pineau, G. Gordon, and S. Thrun. Point-Based Value Iteration: An Anytime Algorithm for POMDPs. *Proceedings of the International Joint Conference on Artificial Intelligence*, 2003.

[Pineau et al. 2003] J. Pineau, G. Gordon, and S. Thrun. Policy-Contingent Abstraction for Robust Robot Control. *Proceedings of Uncertainty in Artificial Intelligence*, 19, pages 477-484, 2003.

[Pineau 2004] J. Pineau. *Tractable Planning Under Uncertainty: Exploiting Structure*. Ph.D. Thesis, Rutgers University, 2004.

[Pineau et al. 2006] J. Pineau, G. Gordon, and S. Thrun. Anytime Pointbased Approximations for Large POMDPs. *Journal of Artificial Intelligence Research*, 27, pages 335–380, 2006

[Png and Pineau 2011] S. Png and J. Pineau. Bayesian Reinforcement Learning for POMDP-Based Dialogue Systems. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 2156-2159, 2011.

[Poupart et al. 2011] P. Poupart, K.E. Kim, and D. Kim. Closing the Gap: Improved Bounds on Optimal POMDP Solutions. *International Conference on Planning and Scheduling*, 2011.

[Precup et al. 1998] D. Precup, R. Sutton, and S. Singh. Theoretical Results on Reinforcement Learning with Temporally Abstract Options. *Machine Learning: ECML-98*, pages 382-393, 1998.

[Puterman 1994] M.L. Puterman. Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., 1994.

[Raol and Gopal 2012] J.R. Raol and A.K. Gopal. *Mobile Intelligent Autonomous Systems*. CRC Press, 2012.

[Rosman and Ramamoorthy 2012] B. Rosman and S. Ramamoorthy. What Good are Actions? Accelerating Learning using Learned Action Priors. *Proceedings of the IEEE International Conference on Development and Learning and Epigenetic Robotics*, 2012.

[Rosman and Ramamoorthy 2012] B. Rosman and S. Ramamoorthy. A Multitask Representation using Reusable Local Policy Templates. *Proceedings of the AAAI Spring Symposium Series on Designing Intelligent Robots: Reintegrating AI*, 2012.

[Rosman and Ramamoorthy 2015] B. Rosman and S. Ramamoorthy. Action Priors for Learning Domain Invariances. *IEEE Transactions on Autonomous Mental Development*, 2015.

[Rosman 2014] B. Rosman. *Learning Domain Abstractions for Long Lived Robots*. Ph.D. Thesis, University of Edinburgh, 2014.

[Roy et al. 1999] N. Roy, W. Burgard, D. Fox, and S. Thrun. Coastal Navigation - Mobile Robot Navigation with Uncertainty in Dynamic Environments. *Proceedings of the IEEE International Conference on Robotics and Automation*, 1, pages 35-40, 1999.

[Roy et al. 2005] N. Roy, G. Gordon, and S. Thrun. Finding Approximate POMDP Solutions Through Belief Compression. *Journal of Artificial Intelligence Research*, 23, pages 1–40, 2005.

[Schmill et al. 2000] M.D. Schmill, T. Oates, and P.R. Cohen. Learning Planning Operators in Real-World, Partially Observable Environments. *Proceedings of the 5th Interantional Conference on Artificial Intelligence Planning System*, pages 246-253, 2000.

[Sherstov and Stone 2005] A.A. Sherstov and P. Stone. Improving Action Selection in MDP's via Knowledge Transfer. *Proceedings of the AAAI*, 5, pages 1024-1029, 2005.

[Silver and Veness 2010] D. Silver and J. Veness. Monte-Carlo Planning in Large POMDPs. *Proceedings of Advances in Neural Information Processing Systems (NIPS-2010)*, pages 2164–2172, 2010

[Smallwood and Sondik 1973] R.D. Smallwood and E.J. Sondik. The Optimal Control of Partially Observable Markov Processes Over a Finite Horizon. *Operations Research*, 21.5, pages 1071-1088, 1973.

[Smith and Simmons 2004] T. Smith and R. Simmons. Heuristic Search Value Iteration for POMDPs. *Proceedings of Uncertainty in Artificial Intelligence*, 2004.

[Smith and Simmons 2005] T. Smith and R. Simmons. Point-Based POMDP algorithms: Improved Analysis and Implementation. *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence*, pages 542-547, 2005.

[Somani et al. 2013] A. Somani, N. Ye, D. Hsu, and W.S. Lee. DESPOT: online POMDP Planning with Regularization. *Advances in Neural Information Processing Systems*, 2013.

[Sondik 1971] E.J. Sondik. *The Optimal Control of Partially Observable Markov Decision Processes*. Ph.D.Tthesis, Stanford University, 1971.

[Sondik 1978] E.J. Sondik. The Optimal Control of Partially Observable Markov Processes Over the Infinite Horizon: Discounted Costs. *Operations Research*, 26, pages 282–304, 1978.

[Spaan and Vlassis 2004] M.T.J. Spaan and N. Vlassis. A Point-Based POMDP Algorithm for Robot Planning. *Proceedings of the IEEE International Conference on Robotics and Automation*, 3, pages 2399–2404, 2004.

[Stolle and Atkeson 2010] M. Stolle and C. Atkeson. Finding and Transferring Policies Using Stored Behaviours. *Autonomous Robots*, 29, pages 169–200, 2010.

[Sutton and Barto 1998] R.S. Sutton, and A.G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[Taylor and Stone 2009] M.E. Taylor and P. Stone. Transfer Learning for Reinforcement Learning Domains: A Survey. *Journal of Machine Learning Research*, 10, pages 1633–1685, 2009.

[Thorpe and Durrant-Whyte 2001] C. Thorpe and H. Durrant-Whyte. Field Robots. *Proceedings of the 10th International Symposium of Robotics Research*, Springer-Verlag, 2001.

[Verma 2005] V. Verma. *Tractable Particle Filters for Robot Fault Diagnosis*. Ph.D. Thesis, Robotics Institute, Carnegie Mellon University, 2005.

[Wiering 1999] M. A. Wiering. *Explorations in Efficient Reinforcement Learning*. Ph.D. Thesis, University of Amsterdam, 1999.

[Wingate et al. 2011] D. Wingate, N.D. Goodman, D.M. Roy, L.P. Kaelbling, and J.B. Tenenbaum. Bayesian Policy Search with Policy Priors. *Proceedings of the International Joint Conference on Artificial Intelligence*, 2011.

[Zhang and Liu 1997] N.L. Zhang and W. Liu. Region-based Approximations for Planning in Stochastic Domains. *Proceedings of the Thirteenth Annual Conference on Uncertainty in Articial Intelligence*, pages 472-480, 1997.