TÉCNICO LISBOA

Reinforcement Learning of Task Plans for Real Robot Systems

Pedro Tomás Mendes Resende

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Examination Committee

Chairperson:	Doctor João Fernando Cardoso Silva Sequeira
Supervisor:	Doctor Pedro Manuel Urbano de Almeida Lima
Co-Supervisor:	Doctor João Vicente Teixeira de Sousa Messias
Member of the Committee:	Doctor Francisco António Chaves Saraiva de Melo

October 2014

Agradecimentos

Em primeiro lugar quero agradecer às pessoas que estiveram envolvidas directamente na tese, nomeadamente ao Prof. Pedro Lima e ao Dr. João Messias. Muito obrigado por todo o apoio que me deram ao longo da tese. Muito obrigado também pela oportunidade de pertencer ao SocRob, que não só me deu conhecimentos e experiência como também uma motivação extra para trabalhar. Agradeço também ao Prof. Rodrigo Ventura, o actual líder da equipa.

Aos meus pais, avós e familiares com que vivo no dia-a-dia, obrigado por me darem todas as possibilidades para tirar o curso e ser capaz de me formar naquilo que gosto. Quero também referir em específico a inspiração que fui retirando já desde os tempos do secundário da Tia Emília. Obrigado pelas explicações e pela motivação que sempre me tentaste passar. Uma palavra especial para a Avó Luz, que apesar de me ter visto a começar o curso, não me vê a acabar. Um beijinho grande.

Aos meus amigos, nomeadamente aos "amigos a sério" e ao quinteto (ou, demasiadas vezes, quarteto) da Praia das Maçãs, muito obrigado por estarem sempre presentes e por serem sempre uma fonte de diversão e descompressão. Espero que assim continuemos durante muito tempo! Um grande abraço e beijinho para todos!

Finalmente, quero agradecer também ao pessoal do SocRob. Aprendi imenso convosco, e nos momentos de lazer nunca me deixaram ficar mal. Foi sempre um prazer trabalhar convosco! Espero que no futuro nos voltemos a cruzar.

Abstract

This thesis focuses on the application of Reinforcement Learning (RL) methods to real robot systems. For that, we create a RL system using the ROS middleware and apply it to a Case Study in the context of domestic robotics.

We start by reviewing the theory of Markov Decision Processes (MDPs) and the existing RL methods. We then review the Markov Decision Making (MDM) ROS package, which provides programming tools for defining tasks as MDPs. Afterwards, we explain how this package was extended with RL capabilities, allowing it to be used for learning.

The application focus of this work is on the SocRob@Home research team, which aims to participate in domestic robotics competitions such as RoCKIn@Home and RoboCup@Home. In this context, we explain how we connect the MDM package with other ROS off-the-shelf packages for task level definition, namely SMACH. We propose a Finite State Machine (FSM) task definition where states can be defined as MDPs using MDM. We also propose the usage of MDM's RL for obtaining policies which can be later used for defining tasks as MDPs. A simulator was created to allow for testing and training for the SocRob@Home robot.

Finally, we present the Case Study. We utilize one of the tasks to be used in RoCKIn@Home Competition 2014 and identify a subtask within it, which contains uncertainty and is also a possible subject for learning, given its environment complexity. The subtask is designed using SMACH with MDM, first by using the RL capabilities to learn the optimal policy and then by using the resulting policy to define the subtask as an MDP within a FSM.

Keywords: Decision Making under Uncertainty, Markov Decision Processes, Reinforcement Learning, Networked Robot Systems, Domestic Robots

Resumo

Esta tese foca-se na aplicação de métodos de Aprendizagem por Reforço (AR) a sistemas de robôs reais. Para isso, é criado um sistema de AR utilizando o "middleware" ROS e é posteriormente aplicado a um caso de estudo.

Em primeiro lugar, a teoria de Processos de Decisão de Markov (MDPs) é revista, bem como os métodos de AR existentes. Depois, é revisto o pacote de ROS Markov Decision Making (MDM), que proporciona ferramentas de programação para definir tarefas como MDPs. Posteriormente, é explicado como este pacote foi melhorado com capacidades de AR, permitindo-lhe ser utilizado para aprendizagem.

O foco de aplicação deste trabalho é na equipa de investigação SocRob@Home, que pretende participar em competições de robótica doméstica como o RoCKIn@Home e o RoboCup@Home. Nesse contexto, é explicado como integramos o MDM com outros pacotes de ROS, disponíveis "chave-na-mão", para definição de tarefas, nomeadamente o SMACH. É proposto o uso de Máquinas de Estados Finitas (MEF) para definição de tarefas onde estados podem ser definidos como MDPs utilizando o MDM. É também proposto o uso das capacidades de AR do MDM para obter políticas que mais tarde podem ser utilizadas para definir tarefas como MDPs. Um simulador foi também criado para permitir testes e treino para o robô do SocRob@Home.

Por último, é apresentado o caso de estudo. É considerada uma das tarefas que vão ser utilizadas no RoCKIn@Home Competition 2014 e é identificada nela uma sub-tarefa que contém incerteza e que é um alvo passível de aprendizagem, dada a complexidade do seu ambiente. A sub-tarefa é projectada utilizando o SMACH com o MDM, primeiro utilizando as capacidades de AR para aprender a política óptima e depois utilizando-a para definir a sub-tarefa como um MDP inserido numa MEF.

Keywords: Tomada de Decisão sobre Incerteza, Processos de Decisão de Markov, Aprendizagem por Reforço, Sistemas de Robôs em Rede, Robôs Domésticos

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Description
AI	Artificial Intelligence
API	Application Programming Interface
Dec-POMDP	Decentralized Partially Observable Markov Decision Process
DP	Dynamic Programming
DT	Decision Theoretic
FSM	Finite State Machine
GPI	Generalized Policy Iteration
MC	Monte Carlo Methods
MDM	Markov Decision Making Package
MDP	Markov Decision Process
POMDP	Partially Observable Markov Decision Process
RL	Reinforcement Learning
ROS	Robot Operating System
SMDP	Semi-Markov Decision Process
TD	Temporal-Difference Learning

Chapter 1

Introduction

1.1 Motivation

Domestic robots, in the form of mobile platforms that interact with networked systems built into the environment, could be a part of our future. In (Gates, 2007), Bill Gates speculates that the current state of robotics could be compared to the state of the computer industry in the mid-1970s, where the idea of a Personal Computer was still very distant.

At the time, computers occupied an entire room and were just used to run back-office operations for major corporations. Following Gates' comparison, notable robotic applications some decades ago were present mostly on industrial process automation (Mandfield, 1989), but also begin to see different environments, such as search and rescue (Casper and Murphy, 2003), human assistance (see MAIS+S[1], MOnarCH[2] and (Martens et al., 2007)), aerospace (see Amazon Prime Air[3]), interplanetary exploration (Maurice et al., 2012), surgery (Davies, 2000), as well as logistics applications (Ibañez-Guzmán et al., 2004). It would then seem that robotic applications are starting to become more widespread.

The field of domestic robots seems to also be evolving, as we begin to see applications such as those of Schiffer et al. (2012), Hirose and Ogawa (2007) and Palacin et al. (2004). Interest in user needs for domestic robots has also begun to be exploited (Sung et al., 2009; Lohse et al., 2008; Dautenhahn et al., 2005).

A brief analysis of these user needs shows much interest in having robots solve our time-consuming drudgeries, such as mowing the lawn, cooking, ironing, etc. It also shows a wish for easy and succinct communication through dialogue. It seems, then, that decision-making capabilities are important in all of these scenarios.

Marvin Minsky wrote in (Minsky, 1982) that "It will be a long time before we learn enough about

[1] `http://gaips.inesc-id.pt/mais-s/index.html`, as of September 2014
[2] `http://monarch-fp7.eu`, as of September 2014
[3] `http://www.amazon.com/b?node=8037720011`, as of September 2014

common sense reasoning to make machines as smart as people are. Today, we already know quite a lot about making useful, specialized, "expert" systems. We still don't know how to make them able to improve themselves in interesting ways.". Today, we still have not improved much in understanding common sense[4], but we have developed some methods that seem to allow robots to decide and improve by themselves.

One of the decision-making methods that were further developed since then is the Markov Decision Processes (MDP) framework. Provided that we can define the problem in terms of its agent and the environment it operates over, as well as a reward function to define good and bad behaviour, the MDP framework allows us to compute the optimal behaviour (in many cases, an approximated optimal behaviour) for the agent. Regarding self-improvement, Reinforcement Learning (RL) removes the necessity of having to define the dynamics of the environment in an MDP, introducing instead the requirement for the agent to explore and experience it. This allows MDPs to be applied to environments with unknown dynamics that possibly change over time. It is also completely self-improving, given its unsupervised nature.

The SocRob@Home project[5] focuses its research in domestic robotics and aims to participate in competitions such as RoCKIn[6] and RoboCup[7]. The work done in this thesis integrates this project and applies decision-making to domestic robots, analyzing its applicability and its benefits when compared to simple deterministic task definition.

1.2 Related Work and State of the Art

The main focus of this thesis relies in applying Reinforcement Learning techniques to real robot systems. These techniques, however, see application in several fields beyond robotics, such as games, economics, operation research, etc. We will start by exploring applications beyond robotics.

In (Moody and Saffell, 2001), methods are presented for optimizing portfolios, asset locations and trading systems based on reinforcement methods. It is explained that using learning methods instead of model-based methods allows for not having to build forecasting models, and improves trading performance by discovering investment policies. They present an adaptive algorithm that they call recurrent reinforcement learning and apply their work to a Case Study on real trades.

Algorithms for playing games have always been application targets of Artificial Intelligence (AI) methods. RL is no exception and has seen application to several games. A well known application of Temporal Difference (TD) learning methods is described in (Tesauro, 1995). Its goal is to have an agent

[4]There have been advances in knowledge representation in the form of Ontology (Russell and Norvig, 2003), but a true understanding of common knowledge is yet to be attained.

[5]`http://socrob.isr.ist.utl.pt/dokuwiki/doku.php?id=start`, as of September 2014

[6]`http://rockinrobotchallenge.eu`, as of September 2014

[7]`http://www.robocupathome.org`, as of September 2014

learn how to play Backgammon through experience. With 30 pieces and 26 possible locations, as well as 20 different ways of playing for a typical dice roll, the number of possible Backgammon positions becomes intractable. To solve this issue, the authors use a neural network to model the problem and use TD techniques to solve it. Another example of RL application to games is (Baxter et al., 1998). In it, the authors present the TDLeaft(λ) method, which combines TD learning with game-tree search for creating an agent to learn to play chess.

RL can also be applied directly to low-level control by defining the state space over control variables. A straightforward SARSA(λ) application for learning the optimal control of a Ball-in-a-cup playing robot arm can be found in (Nemec and Ude, 2011). A more complex application, where Differential Dynamic Programming is used along with inverse Reinforcement Learning to obtain a model from an optimal behaviour can be found in (Abbeel et al., 2007), where the authors successfully design a controller for aerobatic manoeuvres in helicopter flight.

The application of RL that most concerns this thesis is high-level behaviour control. In (Stone et al., 2005), the authors consider the problem of applying RL to soccer robots, more specifically to the keep-away subtask of RoboCup soccer. This subtask involves two teams of robots, and as such has the issues of being a multi-agent system and of including a large space state. An application of a Semi-Markov Decision Process SARSA(λ) algorithm with linear tile-coding function approximation is used with success, given that the agents learned policies that outperformed benchmark policies set by the authors. A different application of RL methods to soccer robots is (Fidelman and Stone, 2004). Here, the goal is to learn a behaviour, grasping the ball, instead of a task. Besides having a different goal, the method for solving the problem is also different. The authors use several algorithms: Policy Gradient, a hill climbing algorithm and downhill simplex (or "amoeba"). Once again, it is reported that the final learned results show improvement over an initial hand-tuned policy.

1.3 Goals

The main purpose of the work developed in this thesis is to create a system for decision-making under uncertainty for the SocRob@Home robot, which will be used in @Home competitions such as RoCKIn.

To reach that goal, several intermediate goals were defined, corresponding to thesis' contributions. The first subgoal was to use the MDM package, which will be described later, as well as to extend it with Reinforcement Learning capabilities, a contribution of this thesis. This extension made it possible to implement a decision-making system in an environment where a model is too complex to determine. The second subgoal was to define the task-level and decision-making system applied to the robot itself, as well as to create a simulator to serve as a basis for testing and learning, which was another contribution of this thesis. Finally, a Case Study was developed, where the proposed method was applied and tested.

An underlying principle throughout the development of the thesis was to use as much off-the-shelf software as possible, as well as using the ROS middleware platform, focusing the contributions on introducing and implementing conceptual level features.

1.4 Thesis Outline

This thesis is organized as follows:

- In Chapter 2, we go over the theoretical background on Markov Decision Processes and on Reinforcement Learning, which corresponds to the basis of the work done in this thesis;
- In Chapter 3, we focus on the MDM package, first by reviewing its capabilities before this thesis, later by explaining the Reinforcement Learning extension that was developed, and finally by presenting the results obtained from testing the implemented system;
- In Chapter 4, we give an overview of the proposed software basis for task-level definition of the SocRob@Home robot. We also explain how the MDM package can be connected with other ROS functionalities and how we use it in our context. Furthermore, we also describe the simulator that was created;
- In Chapter 5, we introduce the Case Study that was developed in the context of this thesis. We use the methods developed in Chapters 3 and 4 and apply it to a real situation;
- Finally, in Chapter 6, we conclude the thesis by summarizing it and delineating possible future work, both in the context of the thesis and of Reinforcement Learning and general task-level learning.

Chapter 2

Background

The work done in this thesis is based on the background of Markov Decision Processes (MDPs) as a framework for modelling and solving problems involving decision making under uncertainty. MDPs were first introduced in (Bellman, 1957a) in the context of Operations Research and nowadays are widely used in the context of single or multi agent decision making.

The end goal of planning for MDPs is to compute the optimal behaviour for an agent within a given environment. A well formed and strict model of the environment is required for using most planning methods for MDPs, which can be a downside due to the fact that in most real applications the true model is complex and hard to define.

Reinforcement Learning (RL) forms a class of solution methods for MDPs that allows for the agent to learn an at least approximated optimal behaviour by experience. There are several different RL methods, which will be explored later, that differ in their learning strategy. Some of those methods do not require a model of the environment, which can be an upside in situations where the environment model is too complex to identify.

Since this thesis focuses on RL, this chapter covers both the subject of MDPs, which are essential for RL, as well as the theory behind RL.

Most of the research done on this subject is based on (Sutton and Barto, 1998) and as such most the notation used follows closely to that used on the book. The brief explanation on Partially Observable MDPs and Decentralized MDPs is based on (Messias, 2014).

2.1 Markov Decision Processes

2.1.1 Basic Concepts

MDPs define an interaction between an agent and the environment where it operates, as represented in Figure 2.1. This interaction depends both on the current state of the environment and on the actions that

the agent can take that possibly change that state. The environment can be modelled as a mapping from states and actions to states that represents the transitions dynamics. A reward model based on a similar mapping is also required and represents either the utility or cost of the agent performing a specific action in a specific state. As such, the MDP model consists of a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ where:

- $\mathcal{S}$ is the state space;
- $\mathcal{A}$ is the action set;
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$ is the transition probability model, where $\mathcal{P}^a_{ss'} = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}$ is the probability of action a taken at state s at time t leading to state s' at time $t+1$;
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the reward model, where $r_t = \mathcal{R}^a_{ss'}$ is the immediate reward of getting to state s' at time $t+1$ from state s taking action a at time t.

The notation $\mathcal{P}^a_{ss'}$ means the probability of reaching state s' after taking action a in state s. The same is valid for different sources, e.g., $\mathcal{R}^a_{ss'}$ means the reward of reaching state s' after taking action a in state s.

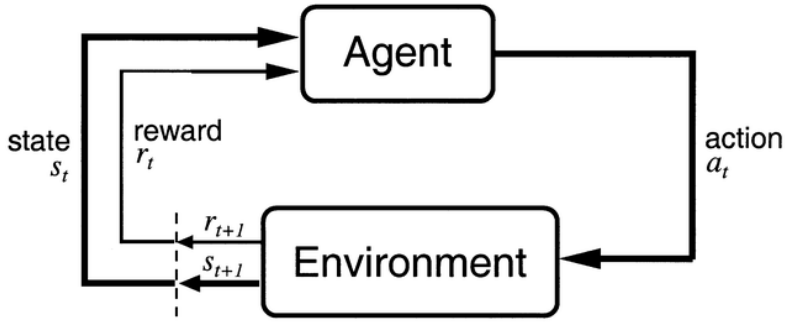


Figure 2.1: MDP Framework. Reprinted from (Sutton and Barto, 1998).

An MDP must satisfy the Markov Property, which means that, at any given decision step, the future state of the system depends only on the present state and action. As an example, knowing the positions of the pieces in a chess game without knowing how they got there is a model that satisfies the Markov Property. Mathematically:

$$Pr\{s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0\} = Pr\{s_{t+1} \mid s_t, a_t\} \tag{2.1}$$

There is a different framework for MDPs where the Markov Property is not required to be strictly satisfied, Semi-MDPs (SMDPs). Despite not being subject of this thesis, further reading on the Semi-MDP framework can be found in (Sutton et al., 1999).

The information presented here and studied for the subject of this thesis only regards single-agent, fully observable MDPs. However, since the MDM package (which will be covered later, in Chapter 3)

encompasses both the multi-agent and the POMDP frameworks, the following paragraphs briefly introduce well-known generalizations of those frameworks.

Partially Observable MDPs are a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{O}, O, \mathcal{R} \rangle$ where $\mathcal{S}, \mathcal{A}, \mathcal{P}$ and $\mathcal{R}$ have the same definition of an MDP and where $\mathcal{O}$ is a set of observations and $O : \mathcal{O} \times \mathcal{S} \times \mathcal{A} \to [0,1]$ is the observation function. Since the state of the system is not directly known in the partially-observable case, POMDPs operate over *belief states* instead of regular states. Those belief states are defined using the observation set and function. More information on POMDPs is found in (Monahan, 1982).

The most general framework for multi-agent MDPs is Decentralized POMDPs (Dec-POMDPs). This framework, besides being defined for the multi-agent case, is also defined for the partially-observable case. The novelty in modelling the multi-agent case is that, in the general case, the agents must reason not only over their actions and observations but also over those of the other agents. For this, the tuple $\langle d, \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{O}, O, \mathcal{R} \rangle$ is defined. The elements $\mathcal{S}, \mathcal{P}, O, \mathcal{R}$ have the same definition as in a POMDP, d is the number of agents, $\mathcal{A} = \prod_{i=1}^{d} \mathcal{A}_i$ is a set of joint actions, where $\mathcal{A}_i$ contains the actions of agent i and, similarly, $\mathcal{O} = \prod_{i=1}^{d} \mathcal{O}_i$ is a set of joint observations, where $\mathcal{O}_i$ contains the observations of agent i. Further reading on Dec-POMDPs can be found in (Bernstein et al., 2002).

2.1.2 Solving MDPs

The goal of solving an MDP is to maximize a target function of the rewards collected by the agent during its execution.

A policy π is a map from states to actions $\pi(s)$, defining a behaviour for the agent. The goal is then to compute a policy that optimizes the behaviour of the agent by analyzing the accumulated reward over a given number of steps. The limit of that number of steps, which can be infinite, is the *horizon*, h. The first step to computing the optimal policy is to compute the optimal utility of each state, which, in the context of MDPs, is computed using the expected discounted reward over future states.

The utility of each state is then represented by a function spanning over every state: the *state-value function* $V^{\pi}(s)$, which is defined in Equation (2.2).

$$V^{\pi}(s) = E_{\pi}\{R_t \mid s_t = s\} = E_{\pi}\left\{ \sum_{k=0}^{h-1} \gamma^k r_{t+k+1} \mid s_t = s \right\} \tag{2.2}$$

The $E_{\pi}\{.\}$ symbol signifies the expected value operator when the agent is following the policy π. This operator is required given that the future states of an MDP cannot be predicted with full certainty due to its stochastic nature, therefore making it necessary to compute the value as an expectation. The γ parameter, $\gamma \in [0,1]$ if $h < \infty$ and $\gamma \in [0,1[$ if $h = \infty$, represents the *discount rate* and determines the present value of future rewards. If γ is closer to 0, immediate rewards weigh more than future rewards

and at the limit, if $\gamma = 0$, the agent chooses the action that maximizes just r_{t+1} (the immediate reward from that action). When using an infinite horizon, γ must be set to lower than 1 to ensure that the series converges. When using a finite horizon, γ is usually 1.

The optimal value function is, then:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \ \forall s \in \mathcal{S} \tag{2.3}$$

and, after applying the *Bellman optimality* equation (Bellman, 1957b),

$$V^*(s) = \max_{a} \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')] \tag{2.4}$$

we reach the equation that, when used iteratively, constitutes the Value Iteration algorithm for solving MDPs:

$$\tilde{V}_{k+1}(s) = \max_{a} \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s')] \tag{2.5}$$

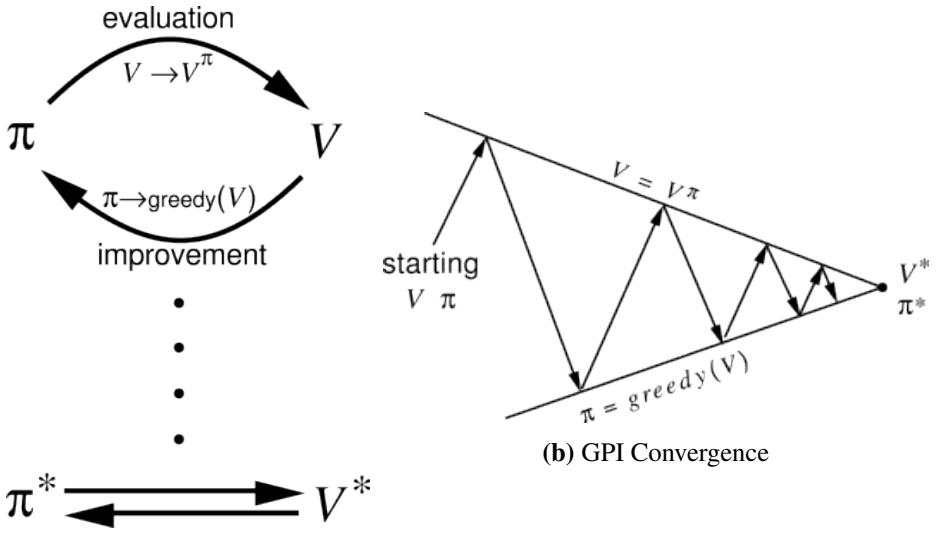
It is proven that, as $k \to \infty$, the value function is guaranteed to converge to an approximation (represented here by $\tilde{V}(s)$) of the optimal value function (Puterman, 1994). After the optimal value function $V^*(s)$ is computed, the optimal policy $\pi^*(s)$ can be generated by simply choosing the most valuable action in each state, i.e., the action associated with the highest expected value.

$$\pi^*(s) = \arg\max_{a} \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')] \tag{2.6}$$

As mentioned, the value iteration algorithm is composed of two different processes. First, there is the iteration process to compute the optimal value function using Equation (2.5) repeatedly. After the optimal value function is obtained, Equation (2.6) is used to compute the optimal policy. An alternative to this method is to intertwine the two processes. In other words, instead of waiting for the first step to end before starting the second one, they can be both active simultaneously.

This alternative is called Generalized Policy Iteration (GPI) (Bertsekas and Tsitsiklis, 1996). The first step is called Policy Evaluation while the second step is called Policy Improvement. The evaluation step refers to computing the value function for the current policy, while the improvement step refers to computing a new policy which is closer to the optimal then the previous version. This algorithm, just like Value Iteration, reaches the optimal value function as well as the optimal policy, but plays an important role in RL methods. Figure 2.2a shows a visualization of the GPI method and Figure 2.2b shows a mockup visualization of the GPI convergence process.

The ideas of Value Iteration or GPI are referred to as the Dynamic Programming (DP) method for



(a) GPI Method

(b) GPI Convergence

Figure 2.2: Generalized Policy Iteration - Visualization. Reprinted from (Sutton and Barto, 1998).

solving MDPs. DP methods suggest breaking apart a complex problem into several simpler and smaller problems, combining their solutions to obtain the solution for the greater problem. Solving an MDP using Value Iteration or GPI uses DP by breaking apart the problem into two subproblems: policy evaluation and policy improvement, combining their results to compute the optimal policy and therefore solving the greater problem.

2.2 Reinforcement Learning

2.2.1 Basic Concepts

As was stated before, RL methods are a class of solutions for MDPs in the sense that they learn the optimal behaviour from experiencing the environment instead of computing it from its model. This means that the model of the environment is not needed in most RL methods. In addition, some RL methods are model-free, meaning that they never come to computing the model (the methods will be explored in detail later).

Given that a model is not available, the value function previously defined in Equation (2.2) becomes insufficient to determine a policy. This is due to the fact that simply looking ahead one step and choosing the action that leads to the best value is not possible without a model. As such, a new value function has to be defined that allows estimating the value of taking each action in each state. The *action-value function* serves that purpose:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t \mid s_t = s, a_t = a\} = E_{\pi}\left\{ \sum_{k=0}^{h-1} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right\} \tag{2.7}$$

Following the same steps that were used with the state-value function, we first reach the optimal

action-value function:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \ \forall s \in \mathcal{S} \ \wedge \ \forall a \in \mathcal{A} \tag{2.8}$$

and, after once again applying the Bellman optimality equation,

$$Q^*(s,a) = \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \max_{\mathrm{a}} Q^*(s',a') \right] \tag{2.9}$$

we reach the final form of the action-value function that is used across all RL methods. Despite still including the model in the form of $\mathcal{P}^a_{ss'}$ in its definition, it will be seen later that the model's knowledge is irrelevant for the update rules for estimating $Q^*(s,a)$ in most algorithms (notably the ones studied here).

Regarding policy improvement using the action-value function, Equation (2.6) is equivalent to:

$$\pi^*(s) = \arg\max_{\mathrm{a}} Q(s,a) \tag{2.10}$$

Another issue that results from not having a model of the environment is the *exploration vs exploitation* trade-off. In order to learn, RL methods must experience the environment. Its exploration must guarantee that the agent tries to perform every action in every state so that it fully experiences and understands what happens in each case. However, if the agent keeps exploring, due to the random nature of exploration, the learnt policy might never converge to the optimal one because the agent may keep exploring a less important set of actions, possibly never even exploring the highest valued set. To mitigate this issue, exploitation, which means repeatedly exploring the highest valued action-state sets, is introduced.

There are some different solutions to balance this trade-off, for instance, ε-*greedy action selection* and *softmax action selection*. We will focus on the first mentioned solution, despite the fact that the second one also provides valuable utility in certain applications.

The ε-greedy action selection method handles the trade-off by introducing a method for selecting which action to perform at any given time. That method sometimes selects an exploiting action, i.e. the action that maximizes its current estimate of Q, while other times selecting an exploring action, i.e. an action that is chosen at random. This selection is random and is based on the ε parameter. With probability ε, a random action is selected, each one with probability $\frac{1}{|\mathcal{A}(s)|}$, while with probability $1-\varepsilon$, a greedy action is selected. Despite being possible to set ε to a constant value, as long as $\varepsilon \in [0,1]$, it is usual to make it a function of time. Setting, for instance, $\varepsilon = 1/t$, has the effect of gradually making the action selection more greedy as time goes by, which leads to exploring the environment early and, when it is already partially explored, starting to exploit it, guaranteeing the policy's convergence.

The exploration vs exploitation issue is studied in-depth in (Thrun, 1992).

2.2.2 Reinforcement Learning Methods

As was the case with planning algorithms for MDPs, RL methods serve the purpose of approximating the optimal policy for MDPs. This subsection explores two different RL methods: Monte Carlo methods and Temporal-Difference methods.

Monte Carlo (MC) methods (Barto and Duff, 1994), require only experience through averaging sample returns, using the ideas of GPI to compute both Q^* and π^*. Although the Policy Improvement segment of GPI carries over unmodified from previously explored DP solution for MDPs, the Policy Evaluation segment must now deal with the fact that there is no longer a model of the environment and that the updates come in the form of experience. There are several different Policy Evaluation methods for MC, for instance *first-visit* MC and *every-visit* MC. Regardless of the chosen method, the general idea is that several visits to the same action-state pair allow for a collection of returns (which in this case are rewards), which can be later averaged to estimate the action-value function. This averaging operation has the inconvenient of requiring a minimum set of visits to the same state-action pair before being possible to perform an update. The update rule for an every-visit MC method is:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_t - V(s_t)] \tag{2.11}$$

where R_t is the average accumulated reward at time t and α is a step-size parameter called the learning rate, $\alpha \in]0,1]$. As was the case with ε, setting α as a function that decreases with time, e.g. $\alpha = 1/t$, guarantees the convergence of the error-correction method in the update rule. To ensure this, the function must respect the following rules: $\sum_{t=1}^{\infty} \alpha(t) = \infty$ and $\sum_{k=1}^{\infty} \alpha^2(t) < \infty$.

Temporal-Difference (TD) learning methods (Sutton, 1988), like MC methods, require only experience and use the ideas of GPI, also carrying over the Policy Improvement method from DP. However, unlike MC, TD methods update estimates based on other learned estimates, without waiting for a final outcome. This nullifies the aforementioned inconvenient of the MC methods, allowing real-time online learning. The update rule for a general TD method is:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.12}$$

where r_{t+1} is the immediately observed reward.

The difference between the two update rules, which allows for TD methods to be online and real-time, is essentially the difference between R_t and r_{t+1}, i.e., the difference in how the experience is collected and evaluated.

Another characteristic that is novel in MC and TD methods when compared to DP, is that they can operate in two different manners: *off-policy* and *on-policy*. The off-policy rule signifies that, given

enough exploration, the agent learns the optimal Q-value function (and implicitly the optimal policy) for the decision-making problem, regardless of the exploration strategy that is used during the learning process. The on-policy rule, on the other hand, signifies that the agent takes its exploratory actions into account while evaluating and optimizing its policy. That is, the learned Q-Value function (and the learned policy) might not be optimal, but they take into account any restrictions that are placed on the behaviour of the agent to promote (or inhibit) exploratory behaviour. These different rules provide different algorithms for MC and TD methods.

We do not explore Monte Carlo methods in-depth, as they were not used in the work done in this thesis, but the next sections explore the two main algorithms for TD methods, SARSA and Q-Learning, respectively on-policy and off-policy learning.

2.2.3 SARSA: On-Policy Temporal Difference Learning

As previously mentioned, the SARSA algorithm (Rummery and Niranjan, 1994) implements the on-policy TD-learning method. The SARSA update rule is:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right] \tag{2.13}$$

Observing the rule, it can be seen that every update uses the tuple $\langle s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}\rangle$, which is where the acronym SARSA is taken from. An algorithm for SARSA is:

Initialize Q(s, a) arbitrarily

while *s not terminal* **do**

- Observe state s
- Choose action a from s using the policy derived from Q (e.g., ε-greedy)
- Take action a and observe the resultant r and s'
- Choose action a' from s' using the policy derived from Q (e.g., ε-greedy)
- $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]$
- Update $s \leftarrow s'$ and $a \leftarrow a'$

end

Algorithm 1: SARSA. Adapted from (Sutton and Barto, 1998).

2.2.4 Q-Learning: Off-Policy Temporal Difference Learning

Q-Learning, first introduced in (Watkins, 1989), is the algorithm that implements the off-policy TD-learning method. The Q-Learning update rule is:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_{\mathrm{a}} Q(s_{t+1},a) - Q(s_t,a_t)\right] \tag{2.14}$$

An algorithm for Q-Learning is:

Initialize Q(s, a) arbitrarily

while *s not terminal* **do**

Observe state s

Choose action a from s using the policy derived from Q (e.g., ε-greedy)

Take action a and observe the resultant r and s'

Choose action a' from s' using the policy derived from Q (e.g., ε-greedy)

$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\,r + \gamma \max_{a'} Q(s',a') - Q(s,a)\,]$

Update $s \leftarrow s'$

end

Algorithm 2: Q-Learning. Adapted from (Sutton and Barto, 1998).

2.2.5 Eligibility Traces

Eligibility traces are a mathematical extension that can be made for TD-learning methods to improve learning efficiency. They were first introduced in (Watkins, 1989) and consist of a temporary record of the frequency of occurrence of an event such as visiting a state or taking an action.

It was mentioned in Subsection 2.2.2 that TD methods differ from MC methods from the fact that in the latter a backup can only be performed when enough reward outcomes are available to perform an average, whereas in the former the immediate reward is enough. Eligibility traces serve as an intermediate measure for this problem. For instance, a two-step backup would be based on the two most recent rewards and on the estimated value of the next two states. This is also valid for three- or four- or *n-step* backups. The n-step reward return is:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{(n-1)} r_{t+n} + \gamma^n V_t(s_{t+n}) \tag{2.15}$$

Applying this backup equation directly still presents the issue of having to wait for n steps before having the complete backup. This is where eligibility traces come in, as a mathematical solution to this issue. The eligibility trace for the state-action pair (s,a), $e_t(s,a)$, is defined as:

$$e_t(s,a) = \begin{cases} \gamma\,\lambda\, e_{t-1}(s,a) + 1 & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\,\lambda\, e_{t-1}(s,a) & \text{otherwise} \end{cases} \qquad \forall s \in \mathcal{S} \wedge \forall a \in \mathcal{A} \tag{2.16}$$

where $\lambda \in [0,1]$, called the *trace-decay parameter*, defines the weight of each backup. The one-step return has a weight of $1-\lambda$, the two-step return has a weight of $(1-\lambda)\lambda$, the three-step return has a weight of $(1-\lambda)\lambda^2$, and so on. On each step, the eligibility traces for all states decay by a factor of $\gamma\lambda$, except for the state that was visited, which increases by 1. Figure 2.3 represents the evolution of an

eligibility trace for a certain state based on when it is visited (the figure regards traces defined just over states and not state-action pairs, but the definition carries over unmodified).

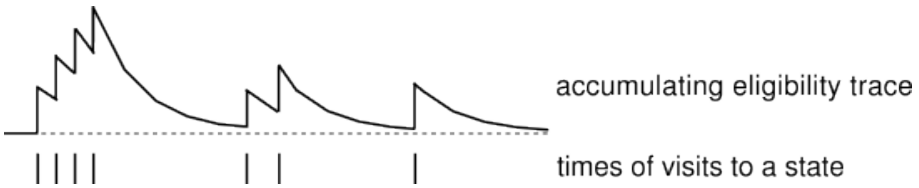


Figure 2.3: Eligibility Traces. Reprinted from (Sutton and Barto, 1998).

The purpose of eligibility traces in the context of RL is to represent how eligible a state-action pair is to be subject of learning changes. Identifying the *TD-error* in the backup equation of the TD algorithms, Equation (2.12),

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \tag{2.17}$$

adapting the backup equation for use with eligibility traces, it becomes (once again, since we are presenting the results for the state-value function, the eligibility traces are defined over states instead of state-action pairs):

$$\Delta V_t(s) = \alpha\, \delta_t\, e_t(s) \;\; \forall s \in \mathcal{S} \tag{2.18}$$

The results in Equations (2.17) and (2.18) are, following the methodology used in Subsection 2.2.2, referent to using the state-value function as the value function. In the next subsections, where the SARSA(λ) and the Q(λ) algorithms are presented, the action-value function will be used instead.

Coming back to the relation between the n-step problem and eligibility traces, we will now analyze the impact of the value of λ. If $\lambda = 0$, all traces are zero except for the trace corresponding to s_t. In this case, the backup equation (2.18) is reduced to original TD-backup (2.12), which means that the eligibility traces have no impact and the method is exactly the same as the original TD. However, if $0 < \lambda < 1$, the values of the preceding states will be changed by a larger amount, but the earlier visited states or state-action pairs are less impacted then the most recent ones. Lastly, if $\lambda = 1$ and $\gamma = 1$, the difference between two consecutive eligibility traces of a given state or state-action pair is zero, and therefore the eligibility traces do not decay at all with time, which implies that the TD method becomes a more general MC method, as the backup equation (2.18) becomes equivalent to the MC backup equation (2.11).

When using eligibility traces, TD methods are called TD(λ). So, summarizing the last paragraph, the Q-Learning and SARSA algorithms from Subsections 2.2.4 and 2.2.3, respectively, are TD(0) algorithms. Also, MC methods are TD(1) algorithms. The next subsections present the general SARSA(λ) and Q(λ) algorithms.

A variation of eligibility traces, called *replacing traces*, shows increased performance in some cases. Their utilization within the RL framework is the same as eligibility traces, as only their mathematical definition changes. As replacing traces are not used in this thesis, the reader should refer to (Singh and Sutton, 1996) for an in depth-explanation of the subject.

2.2.6 SARSA(λ)

The adaptation of the SARSA algorithm with eligibility traces was first introduced in (Rummery and Niranjan, 1994). Its update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \delta_t e_t(s_t, a_t) \tag{2.19}$$

where

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{2.20}$$

An algorithm for SARSA(λ) is:

Initialize Q(s, a) arbitrarily and e(s, a) = 0 $\forall s \in \mathcal{S} \wedge a \in \mathcal{A}$
while *s not terminal* **do**
  Observe state s
  Choose action a from s using the policy derived from Q (e.g., ε-greedy)
  Take action a and observe the resultant r and s'
  Choose action a' from s' using the policy derived from Q (e.g., ε-greedy)
  $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
  $e(s, a) \leftarrow e(s, a) + 1$
  forall the *(s, a) pairs* **do**
    $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$
    $e(s, a) \leftarrow \gamma \lambda e(s, a)$
    Update $s \leftarrow s'$ and $a \leftarrow a'$
  end
end

Algorithm 3: SARSA(λ). Adapted from (Sutton and Barto, 1998).

2.2.7 Q(λ)

There are two different versions of the Q(λ) algorithm, Watkin's Q(λ), (Watkins, 1989), and Peng's Q(λ), (Peng and Williams, 1996). The work done in this thesis follows Watkin's version of the algorithm, and from here on out, references to Q(λ) always refer to it.

Due to the fact that Q-Learning is an off-policy method, special care is required when adapting it to use eligibility traces. To avoid introducing errors in the learning policy that arise from selecting

both exploratory and greedy actions from the following policy, Q(λ) only looks ahead as far as the last exploratory action. This comes in contrast with SARSA(λ), which looks ahead until the end of the episode. To handle this change, the previous definition of eligibility traces, in Equation (2.16), must be modified. The new trace update can be divided into two steps. First, the trace of all state-action pairs are decayed by $\gamma\lambda$, or, if an exploratory action has been taken, set to 0. Then, the trace corresponding to the current state-action pair is incremented by one. The resulting trace is:

$$e_t(s,a) = \mathcal{I}_{ss_t}\mathcal{I}_{aa_t} + \begin{cases} \gamma\lambda\, e_{t-1}(s,a) & \text{if } Q_{t-1}(s_t,a_t) = \max\limits_{a} Q_{t-1}(s_t,a) \\ 0 & \text{otherwise} \end{cases} \quad \forall s \in \mathcal{S} \wedge a \in \mathcal{A} \tag{2.21}$$

where $\mathcal{I}_{xy}$ is equal to 1 if $x = y$ and 0 otherwise.

Using the new traces equation, the update rule is:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\delta_t\, e_t(s,a) \tag{2.22}$$

where

$$\delta_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t) \tag{2.23}$$

An algorithm for Q(λ) is:

Initialize Q(s, a) arbitrarily and e(s, a) = 0 $\forall s \in \mathcal{S} \wedge a \in \mathcal{A}$
while *s not terminal* **do**
  Observe state s
  Choose action a from s using the policy derived from Q (e.g., ε-greedy)
  Take action a and observe the resultant r and s'
  Choose action a' from s' using the policy derived from Q (e.g., ε-greedy)
  $a^* \leftarrow \arg\max_b Q(s', b)$
  $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s, a)$
  $e(s, a) \leftarrow e(s, a) + 1$
  forall the *(s, a) pairs* **do**
    $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$
    if $a' = a^*$ **then**
      $e(s, a) \leftarrow \gamma \lambda e(s, a)$
    end
    else
      $e(s, a) \leftarrow 0$
    end
    Update $s \leftarrow s'$ and $a \leftarrow a'$
  end
end

Algorithm 4: Q(λ). Adapted from (Sutton and Barto, 1998).

Chapter 3

A Framework for Reinforcement Learning in Real Robot Systems

One of the goals of this thesis is to extend the previously mentioned Markov Decision Making (MDM) ROS package with Reinforcement Learning capabilities. The MDM package, (Messias, 2014), comprises a software basis for decision-theoretic (DT) methods such as MDPs and POMDPs. It allows the user to define abstract actions and states based on perceptions, that later are used by a controller that implements the DT method. It does not, however, solve the MDP problem, only accepting a previously computed policy. The package, can be easily connected with other ROS packages, and therefore can also be easily integrated into a real robot system.

The goal of extending the package with RL capabilities carries over two ideas from the initial goals of MDM: being easy to use and providing flexibility to the user. Moreover, to provide as much utility as possible, the RL extension includes both SARSA and Q-Learning, as well as their eligibility traces adaptations.

In this chapter we first go over the previous functionalities of the MDM package, later going over the RL extension. The initial description of the MDM package is based on (Messias, 2014), which contains in-depth usage examples in its Appendix B. We assume that the reader is familiar with ROS concepts such as nodes, topics, services and the *actionlib* interface. Information on these and other ROS related concepts can be found in (Quigley et al., 2009).

The code for the MDM package is available in a Git repository[1], as well as in the ROS repository[2] for public download and usage. A tutorial and example of implementation of the RL extension, to be added later to the one already available MDM tutorial, is presented in Appendix C.

[1] `https://github.com/larsys/markov_decision_making`, as of September 2014
[2] `http://wiki.ros.org/markov_decision_making`, as of September 2014

3.1 The MDM Package

The package is organized in *layers*. Each layer is a ROS node and corresponds to a certain functionality. In its initial state, the package had four layers: the *state layer*, the *observation layer*, the *action layer* and the *control layer*. These layers have to be instantiated and configured by the user. Besides these main components of the package, there are two sub-components; the *Predicate Manager* and the *Topological Tools* which provide auxiliary tools for the problem definition. Each layer and component serves a specific purpose:

- **State Layer:** where a discrete set of states describing the world, based on predicates, are defined;
- **Observation Layer:** provides the observation functionalities for POMDPs;
- **Action Layer:** used to define abstract actions, which are bound to a user defined callback function;
- **Control Layer:** where the implemented policy is interpreted;
- **Predicate Manager:** where predicates are defined;
- **Topological Tools:** provides functionality for topological-based navigation, abstracting the map as a labeled graph, thus allowing for defining simple navigation actions such as ”Move Up”.

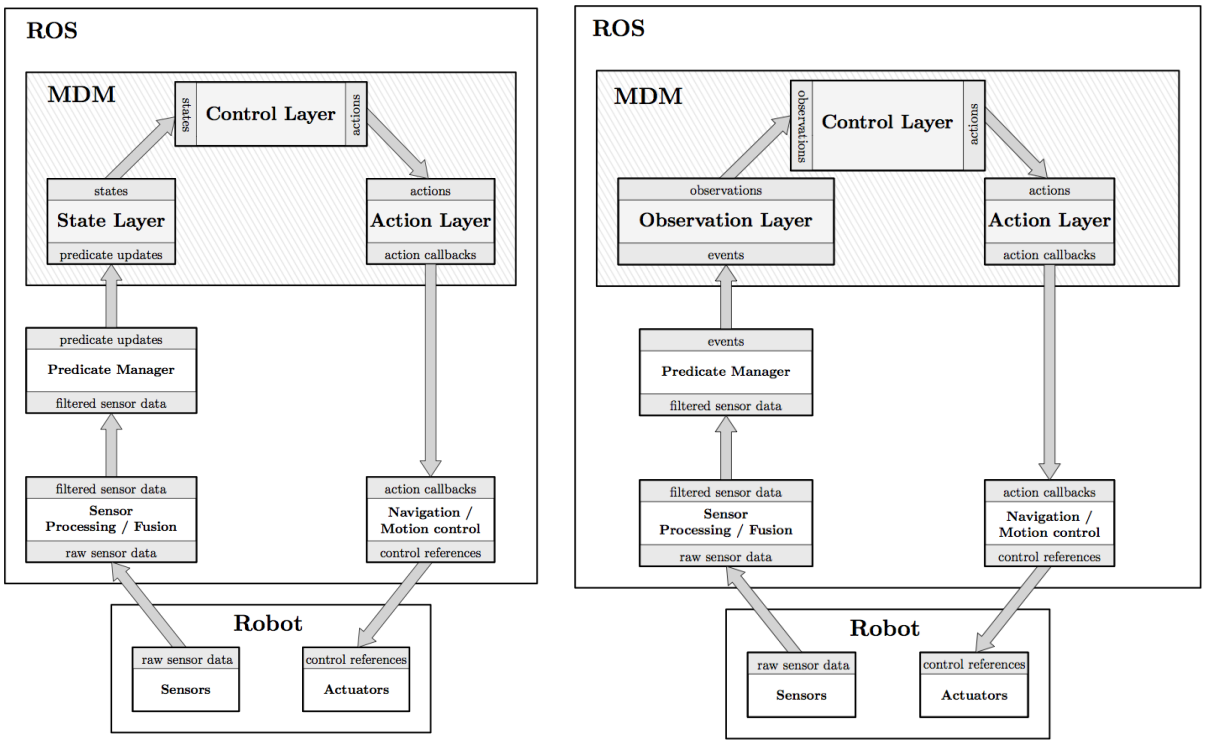
Figure 3.1 shows an example integration of the MDM package and other ROS functionalities. In it, we can see the control loop between the agent and the environment, wherein the state layer receives information about the environment from the Predicate Manager, forwarding the state to the control layer. Afterwards, according to the policy defined in it, the corresponding action is selected from that state, sending it to an ”executor” node, which interprets the action and sends it to the actuators.

3.1.1 The State Layer

The functionality of the state layer is to interpret logical predicates into integer-valued state representations. To define predicates, a simple API is provided by the Predicate Manager component.

There are two types of predicates that can be defined in the Predicate Manager: general abstract predicates and topological predicates. The former allows for definition based either on sensory information or through propositional calculus with other predicates. The latter uses the Topological Tools component, defining a predicate based on whether or not the robot is in a certain labeled area.

For instance, one could define a sensorial-based predicate, *IsSpeaking*, which is true when the robot is speaking, and a topological-based predicate *IsInKitchen*, which is true when the robot is in the map area labeled as kitchen. A third predicate can then be defined over these two predicates, based on propositional calculus. It could be, for instance, $IsSpeaking \wedge IsInKitchen$, $IsSpeaking \vee IsInKitchen$, etc.



(a) MDM Control Loop for MDPs.

(b) MDM Control Loop for POMDPs.

Figure 3.1: MDM Control Loop. Reprinted from (Messias, 2014).

Using the built-in predicates defined, the Predicate Manager publishes a list of updates whenever one or more predicates change their value. Using this information, the state layer maps predicates into state factors $\mathcal{X} = \{\mathcal{X}_i : i \in \{1,...,k\}\}$, which in term map states $\mathcal{S} = \Pi_{i=1}^{k} \mathcal{X}_i$. This mapping can be defined by the user in one of two possible different ways. The first way is to bind a state factor directly to a predicate, meaning that the state factor will always have the same value that the predicate does. The second way is to bind a state factor to a set of mutually exclusive predicates under the condition that only one of those predicates is true at any given time.

Whenever the state layer receives a predicate update, it publishes an integer value which corresponds univocally to a certain configuration of state factors. This publication is made through the */state* topic.

3.1.2 The Observation Layer

When using a POMDP controller, the observation layer replaces the state layer, as can be observed by comparing both diagrams in Figure 3.1.

The difference between the state and observation layers is that, while in the fully observable case states are mapped directly from predicates, in the partially observable case observations are mapped from instantaneous occurrences. In other words, states are defined over persistent conditional values, while observations are defined over instantaneous conditional changes.

The Predicate Manager tool also allows the definition of named events. These can be defined either

over a propositional formula of predicates or over a specific source of data. The former triggers an event whenever the formula becomes true, and the latter triggers based on a user defined condition over the data (e.g., when temperature rises above a certain value).

As was the case with states, the observation space can also be factored and, when an observation occurs, the observation layer publishes it as an integer joint value which corresponds univocally to the current observations. The publication is made through the */observation* topic.

3.1.3 The Control Layer

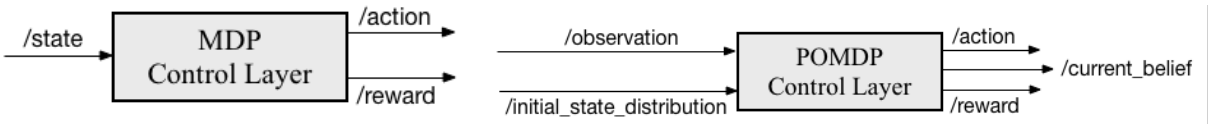
The control layer serves the purpose of interpreting the policy defined within it, based on the state published by the state layer, outputting the corresponding action.

There are implementations of the control layer for both MDPs and POMDPs, and the user defines which one he wishes to use. As mentioned before, the MDM package does not solve the MDP or POMDP problems itself, accepting instead a previously computed policy.

Another characteristic of the controller to be chosen by the user is whether the control method is time-based or event-based, i.e., whether the control is performed on a time interval basis or upon the perception of a new state.

As seen in Figure 3.2a, the MDP controller listens to the */state* topic, publishing to the */action* topic and, optionally, the most recently received reward to the */reward* topic.

As for POMDP controllers, which communication diagram can be seen in Figure 3.2b, the topics listened to are */observation* and */initial_state_distribution*. The latter of those can be used to set the initial belief of the POMDP. Additionally, they publish to the */action*, */reward* and */current_belief* (which publishes the belief state at run-time) topics.



(a) MDP Controller Communication Diagram.

(b) POMDP Controller Communication Diagram.

Figure 3.2: Control Layer Communication Diagram.

All controllers can be started or stopped at run-time, allowing the execution of a controller to be abstracted as an action. This allows for hierarchical relations between MDM processes, a topic that will be further explored later.

3.1.4 The Action Layer

The main goal of the action layer is to interpret the action requests published by the control layer and to pass them over to other ROS modules for their execution. This is achieved by univocally assigning a callback function to each action received by the control layer.

Returning to the topic of hierarchy, the action layer makes it possible to abstract MDM controllers as actions, allowing the user to implement hierarchical dependencies between MDPs and POMDPs. An illustration of using this hierarchy functionality is shown in Figure 3.3.

The action layer does not publish any information, solely subscribing to the */action* topic to gather which action the controller publishes.

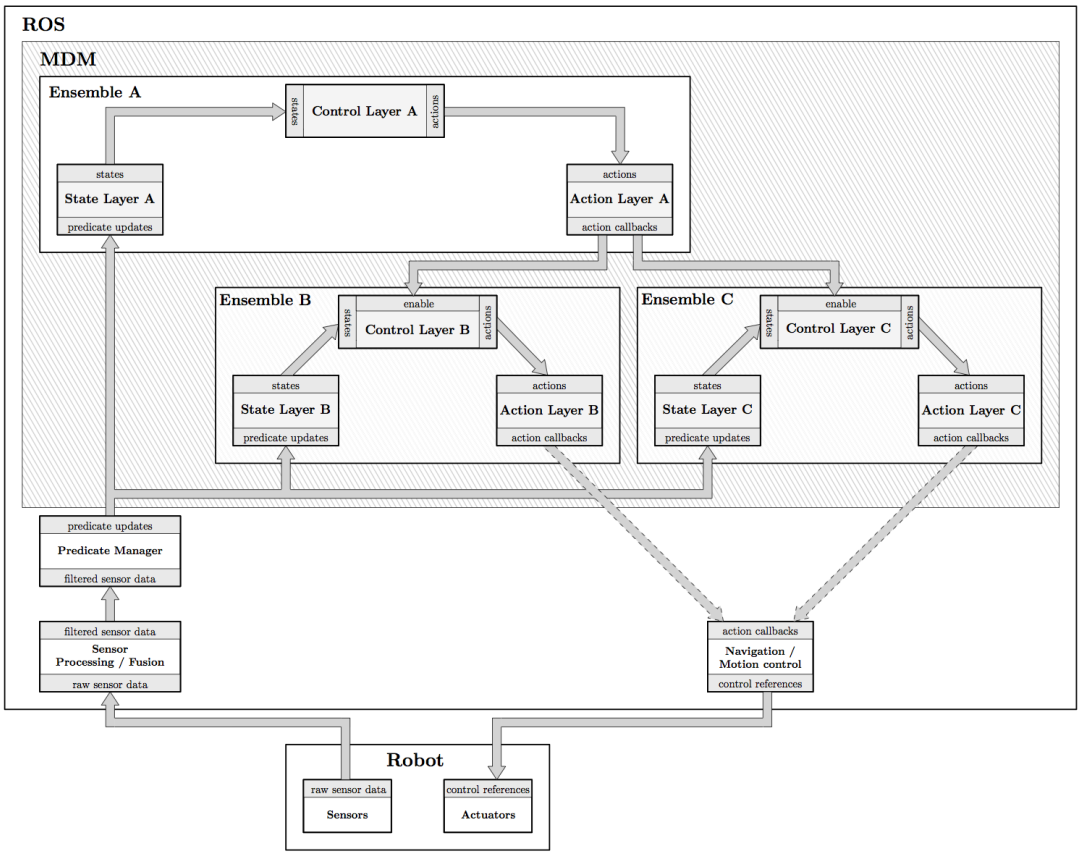


Figure 3.3: MDM Hierarchy. Reprinted from (Messias, 2014).

3.2 Extension for Reinforcement Learning

The desired usage method for the MDM package with the RL extension is for the user to first define the problem in the action and state layers, later using its learning capabilities to obtain the approximated optimal policy for it. Afterwards, the user can use the computed policy to along with the planning capabilities of the package to run and apply the learned task.

The goal of the RL extension to MDM is to provide the following features:

- Directed at task-plan learning;
- Supports both Q(λ) and SARSA(λ) agents, and is structured so that additional algorithms are easily implemented;
- Provides real-time introspection on the learning process;
- Allows offline and online learning;
- The planning capabilities of MDM can be used to execute the learned task.

To be able to accomplish these desired features, the development of the following subgoals is required:

- Maintain the MDM structure, mainly in terms of its layered architecture;
- Provide a simple way for the user to implement the learning system;
- Implement the SARSA and Q-Learning algorithms, as well as their eligibility traces counterparts;
- Implement the ε-greedy action selection method;
- Create a system where the parameters can be set either as constants or functions of time;
- Provide a real-time introspection tool, namely a GUI;
- Give flexibility for future improvements on the package.

3.2.1 The Learning Layer

The goal of the learning layer is to provide an interface for the user to run RL algorithms over the problem defined in the state and action layers. To achieve this, seamless integration with the rest of the MDM package is required. As such, we maintain the layered structure of the MDM package, and developed a new layer for the RL extension, named the *learning layer*. This new layer is related to the control layer as the state layer is related to the observation layer, i.e., it serves as a substitute for it when learning is required instead of control. Figure 3.4 shows the integration of the learning layer with the other layers. Besides the new layer, the ε-greedy action selection method was also implemented. For the user, defining a RL problem with MDM is the same as defining an MDP/POMDP problem, with the difference that instead of implementing a control layer, the user implements a learning layer.

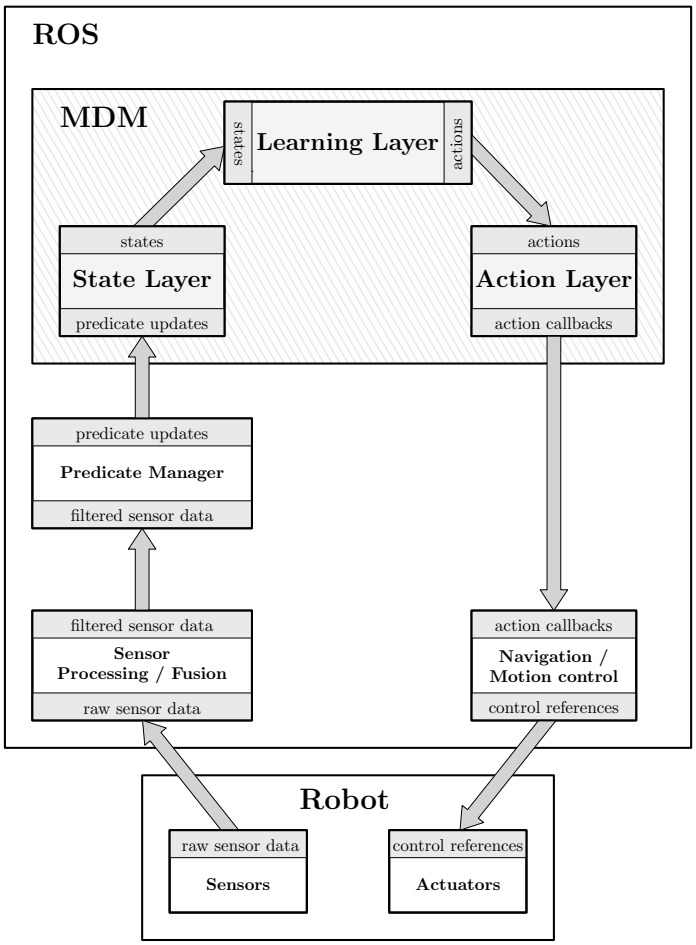


Figure 3.4: Learning Layer Integration.

Given that, for learning, control is also required, the learning layer contains an implementation of a control layer. This means that, while taking care of the RL algorithms, the learning layer also provides the controller with a possibly ever-changing MDP problem.

The implemented algorithms, namely SARSA and Q-Learning, are built upon this layer. In terms of code, it follows a hierarchy in which the algorithm classes are created over the learning layer class. This structure in shown in Figure 3.5.

In terms of communication with the other components, according to what was previously stated about seamless integration, the learning layer communicates with the state and action layers in exactly the same way that the control layer does. A diagram is shown in Figure 3.6.

3.2.2 Parameters

There are eight different parameters to be set by the user: α (see Subsection 2.2.2), λ (see Subsection 2.2.5), γ (see Subsection 2.1.2), ε (see Subsection 2.2.1), the controller type, the reward model, the reward for impossible actions (this topic will be covered later, in Subsection 3.2.4) and the policy update frequency. The first four of these are parameters used by the RL algorithm. The other four influence

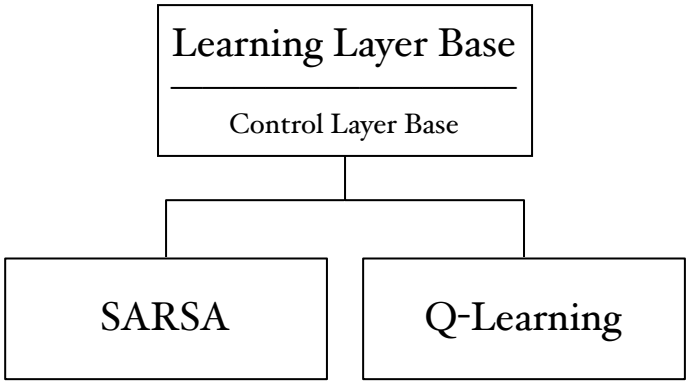


Figure 3.5: Learning Layer Class Structure. The SARSA and Q-Learning classes are built on top of the Learning Layer Base class, which in turn contains an instance of the Control Layer Base class.

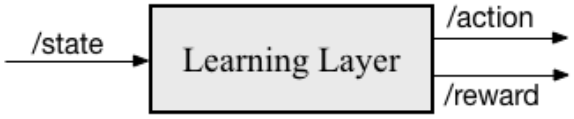


Figure 3.6: Learning Layer Communication Diagram.

the behaviour of the MDM system. The controller type refers to whether the controller should make decisions on a time basis or on an event basis (where an event is a change in state). The reward model is used to set the user's reward definition, i.e., whether it is defined as a map from states to rewards ($\mathcal{R} : \mathcal{S} \rightarrow \mathcal{R}$) or as a map from state-action pairs to rewards ($\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{R}$). The reward for impossible actions is the reward that is given to the agent whenever an action is requested to be performed in a state where it is not feasible. This issue will be further explored later. Finally, the policy update frequency defines the amount of decision episodes in-between policy updates, saving the policy to file, saving the Q-Values $Q(s,a)$ table to file and saving the eligibility traces $e(s,a)$ table to file.

Given that both α and ε can be defined as functions of time, we give the option of either having them as constants or as functions. The α parameter has three different types: constant, $1/t$ and $1/t^2$. The ε parameter has five different types: constant, $1/t$, $1/t^2$, $1/\sqrt{t}$ and e^{-Ct}. The reasoning for these ε types is given in the ε-greedy section of this chapter.

The controller type, α type and ε type are defined when instantiating the Learning Layer class. The other parameters, λ, γ, the reward model, the reward for impossible actions and the policy update frequency are implemented as ROS parameters, using its parameter server[3]. If the α and ε types are set as constant, MDM also gets their value through the parameter server.

All of the parameters have default values that are used when the user does not set them himself. This is especially important regarding λ, as it defines whether eligibility traces are used in the algorithms. The default value for λ is zero, defaulting the algorithms to their basic TD(0) implementation.

[3] `http://wiki.ros.org/Parameter%20Server`, as of September 2014

3.2.3 ε-greedy Action Selection Method

The ε-greedy action selection method is implemented over the previously existing deterministic policy. The added functionalities are an update function, a get action function and a save the policy to file function.

The update function serves the purpose of updating the policy according to Equation (2.10),

$$\pi^*(s) = \arg\max_{\mathrm{a}} Q(s,a)$$

The function for getting an action uses the ε-greedy methodology described in Subsection 2.2.1, where the first step is to update ε's value, unless it is set to constant. Saving the policy a to file allows the user to use it with the planning capabilities of MDM after the learning process. There is also the functionality to save the $Q(s,a)$ function to a file, which enables starting a learning episode with initial conditions set to the final conditions of a previous episode. This allows for faster learning, as well as guaranteeing that progress is not lost if something goes wrong.

An instance of the ε-greedy class is always created whenever a learning layer is instantiated, given that it is the methodology used for exploration.

Regarding the ε functions mentioned in Subsection 3.2.2, they were added due to the fact that the most common $\varepsilon = 1/t$ function was, in our experience, not exploring the environment fully before starting to exploit it. To overcome this issue, we tried to find a function that obeyed to the requirements of the ε-greedy method and that decreased slower with time then $1/t$. There are only two requirements for the function: that $\varepsilon \in [0,1]\ \forall\, t$, which allows for ε to be used as a probability, and that $\lim_{t\to\infty} \varepsilon(t) = 0$, which guarantees that the policy becomes greedy over time.

We explored two eligible functions that converge slower to zero than $1/t$ which are shown in the plot in Figure 3.7. These functions are $1/\sqrt{t}$, which is represented in green, and e^{-Ct}, which is represented in blue. For comparison, the $1/t$ function is represented in red.

Inspecting the plot, it can be observed that both the $1/t$ and $1/\sqrt{t}$ functions stop exploring sooner than e^{-Ct}, and given the additional benefit that the constant C allows for managing the function's behaviour to adapt it to different scenarios, we decided to implement as a function for ε. In the plot presented here, $C = 1/50$.

3.2.4 Dealing with Impossible Actions

Given the implementation of the ε-greedy method and the way that the action and state layers are built in MDM, when an action is chosen from the policy, it may be impossible to realize in the current state.

Using the example of a navigation task, let us imagine a situation where there are two rooms con-

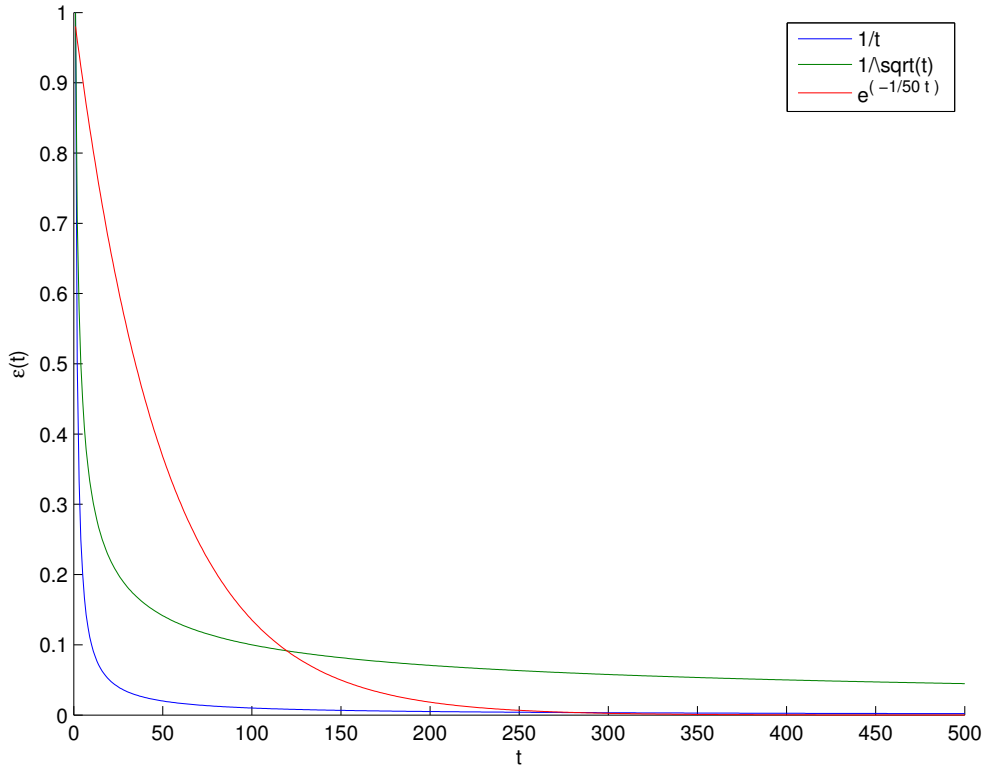


Figure 3.7: Possible $\varepsilon(t)$ Functions.

nected with each other. There are two states: room one, which corresponds to state s_1, and room two, which corresponds to state s_2. There are also two actions: go right, which corresponds to action a_1, and go left, which corresponds to action a_2. From s_1 it is possible to reach s_2 through action a_1, and, conversely, from s_2 it is possible to reach s_1 through action a_2. In practical terms, when using MDM to build this scenario in a real robot system, the effect of executing a_2 in s_1 or a_1 in s_2, which are physically impossible to realize, is not well defined.

Since the ε-greedy action selection method involves choosing random actions, this becomes an issue when using the RL algorithms, because these impossible or ill-defined actions may be selected. As such, an optional system was developed to facilitate generating a new action when the previously tried one was impossible to realize. This system uses a ROS service[4] that should be called whenever the action layer detects an impossible action. When it is called, MDM assumes that the action was executed and merely led to the same state, which means that a backup for that transition will still be used by the algorithm. The reasoning behind the decision to have the backup count, is that impossible actions should be penalized, given that they are most likely incorrect. To provide flexibility, we have created a parameter which allows the user to define a preset reward to be given to the agent when it tries to perform impossible actions. This releases the cumbersome need of modifying the reward model by hand to include such rewards.

[4] `http://wiki.ros.org/Services`, as of September 2014

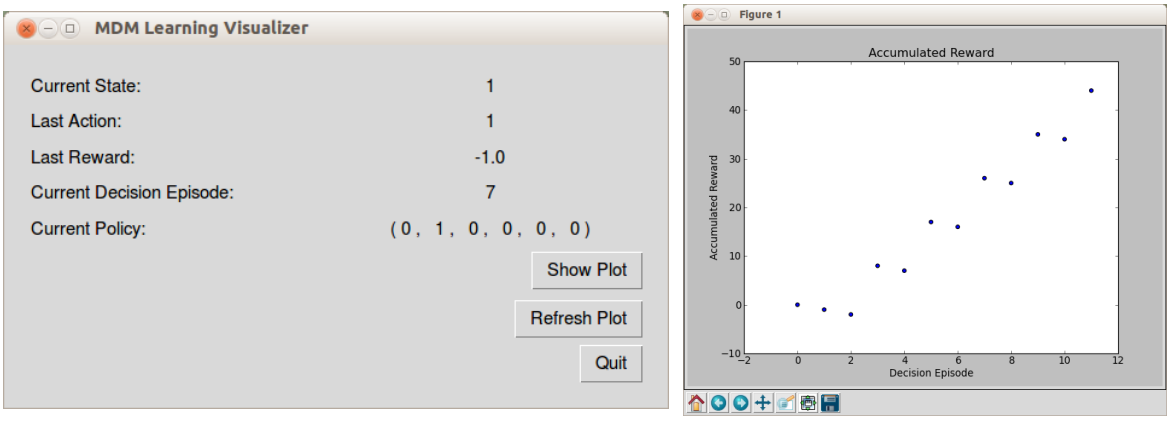
The use of state-conditional action spaces, i.e., action spaces that are defined based on specific state conditions, are a possible alternative to our solution. In that case, the actions possible for each state are already known, and as such, diverting from the model would not be required. The downside of this alternative is that it requires the user to specify the pre-conditions for each action, which requires more implementation work than our solution. Additionally, in some cases, it might not be known which actions are possible to perform in each states, especially in the context of robotics.

The Topological Tools functionality already has this service call built in due to the fact that impossible actions are obvious to identify in such a system (given that the user specifies the topological connections, a topological node that does not have a connection to any other topological node through a specific navigation action, makes that action impossible to realize in that node).

3.2.5 The GUI and Logger Functionalities

To give the user a basis for debugging and analysis, two extra functionalities were created, a GUI to be used in real-time and a logger for posterior inspection.

The GUI subscribes to the state, action, reward and policy topics, displaying the information in a simple grid view. A plot of accumulated reward is also available. Figure 3.8a shows the GUI running, while Figure 3.8b shows the plot.



(a) MDM GUI.

(b) MDM GUI Reward Plot.

Figure 3.8: MDM GUI Functionality.

To provide a method for posterior inspection of the learning progress, the logger functionality subscribes to the same topics as the GUI, organizing and saving the information received to a file. The

following text box contains an excerpt of an example log.

```
At Decision Episode 13
        State -> 3
        Action -> 1
        Reward -> -1.0
        Policy -> ( 0 , 0 , 0 , 1 , 0 , 0 )
At Decision Episode 14
        State -> 2
        Action -> 0
        Reward -> -1.0
        Policy -> ( 0 , 0 , 0 , 1 , 2 , 0 )
At Decision Episode 15
        State -> 3
        Action -> 1
        Reward -> 100.0
        Policy -> ( 0 , 0 , 0 , 1 , 2 , 0 )
```

3.2.6 SARSA and Q-Learning

Both SARSA and Q-Learning are implemented as classes built on top of the *LearningLayerBase* class, which is the implementation of the Learning Layer, and are implemented based on the algorithms presented in Section 2.2. These are the classes that the user instantiates for using MDM's RL. As was mentioned before, these classes also instantiate a controller that handles the control segment of learning. When using the classes, the user must specify either three or four arguments. Both SARSA and Q-Learning require the path to the policy file, to the reward file and to the Q-Values file. Optionally, the user can also provide a path to a file containing the eligibility traces values. All of these files correspond to Boost's[5] matrix/vector default configuration and contain the values of the respective function.

The *LearningLayerBase* class contains a Q-Values table, which is always initialized as zeros, except when the file that contains them is not empty when the program is initiated. If it is not empty, the file is loaded into the table. The same principles are used with the eligibility traces table. If λ is set to zero, the TD algorithms SARSA(0) or Q(0) are used. Otherwise, the methods with the eligibility traces extension SARSA(λ) or Q(λ) are used.

The methods operate when a new state is received through the */state* topic. Upon acquisition of a new state, the most recent action taken and the most recent reward received are gotten from the controller. When enough information is collected to perform a backup, the Q-Values are updated. Afterwards, the policy is updated (according to Subsection 3.2.3), published to a ROS topic and saved to a file each policy update frequency decision episodes. The Q-Values table, and, if applicable, the eligibility traces table, are also saved to a file according to the policy update frequency parameter.

[5] `http://www.boost.org`, as of September 2014

Finally, both methods have a ROS service advertiser for the republish service explained in Subsection 3.2.4. As previously described, the user can use this service by calling it from the action layer.

3.2.7 MDM Extendability

We mentioned in the introduction to this chapter that one the ideas carried over from the original MDM iteration was to provide as much flexibility to the user as possible. By this, we mean providing ways for the user to adapt the package to his needs, extending it with extra functionalities as required. As such, we used a class hierarchy that allows for creating additional algorithms, as well as broad and adaptable definitions of policy and parameters.

The base class *LearningLayerBase* contains both the information and the functions common to both SARSA and Q-Learning. In principle, this allows for implementing other methods on top of the same base class and alongside the already implemented ones.

Implementing an alternative action selection method, e.g. softmax, is possible to do alongside the ε-greedy class. Implementing stochastic policies can also be done on the base level, substituting the deterministic policy.

Creating different functions for the parameters is also indifferent to the rest of the code. Each parameter, as mentioned before, has different "types". Besides the constant type, which behaves differently, every other type contains an update function, which serves the purpose of updating the parameter's value according to it. Creating a new function just requires creating a new type and a new update function. The rest of the code already calls these update functions when required, and as such a new one is indifferent to the rest of the code.

3.2.8 Considerations on Partial Observability, on Multi-Agent Decision Making and on Hierarchical Reinforcement Learning

Since the planning capabilities of MDM extend to the multi-agent and the partially observable frameworks as well as to hierarchical structures of (PO)MDPs, we decided to explain how the RL capabilities deal with those cases.

There have been several different approaches in the past to apply RL methods to POMDPs. However, it has been shown (Singh et al., 1994) that the conventional RL framework is inadequate to deal with partial observability. Several methods have been proposed to solve this issue (Kaelbling et al., 1996), but, as far as we know, it is still an open problem. Although it is possible do use MDM's RL functionalities over a problem defined as a POMDP, it is still, from the perspective of the agent, a non-Markovian problem and, as such, the RL methods are not guaranteed to converge.

Multi-agent RL refers to having multiple agents with interacting or competing goals. It requires that,

to each agent, other agents are not represented by being part of the environment, but rather as agents that are capable of influencing it. (Tan, 1993) studies the difference between independent and cooperative agents in multi-agent RL. In the MDM case, it is possible to have RL running on several agents that are operating at the same time. However, given that the development of RL was built on top of the basic MDP framework, multi-agent RL as defined in (Claus and Boutilier, 1998), or in (Littman, 1994) is currently not possible.

Several methods for hierarchical RL have been proposed, such as Options, (Sutton et al., 1999), Hierarchies of State Machines, (Parr and Russell, 1998) and MAXQ Value Function Decomposition, (Dietterich, 2000). These methods have been reviewed in (Barto and Mahadevan, 2003). Regardless of the specific method, hierarchy in RL means having a hierarchy of MDPs, where, while learning, information can be passed through bottom-to-top. This means that, when an action from the topmost MDP initiates a lower MDP in the hierarchy, some information on the backups performed in the second MDP must be reflected in the value function of the first one. This process is non-Markovian, meaning that most methods present the problem as an SMDP.

Regarding the MDM case, implementing learning hierarchically means implementing a system similar to the one in Figure 3.3, except using learning layers instead of control layers (or even a mixture of both). Let us imagine such an implementation in which there are two MDPs, where the second can be "launched" by the first through an action. That action has a reward value associated with it. Imagine then that the action is called and that the second MDP is put through a learning episode. When that episode is complete, the policy of the second MDP is different than it was before the episode, and the first MDP is un-paused. However, the only information that the first MDP receives from this process is the reward from executing the action, which means that no information from the learning episode of the second MDP is transmitted to the first one.

As such, it is possible to create a hierarchical structure of MDPs and even RL processes, but true hierarchical RL would require changes in MDM's communication structure.

3.3 Testing

To test the implemented system and algorithms, a simple scenario was built within a simulator with the goal of providing empirical working proof. The simulator that was used is ROS Stage[6], which simulates navigation in a 2D world. We will now describe the scenario and present the obtained results.

[6] `http://wiki.ros.org/stage`, as of September 2014

3.3.1 The Testing Environment

The goal of the testing environment is to be a simple and straight forward scenario, with results that are easy to analyze and interpret but that concern nonetheless a typical robotic task. As such, we designed a simple navigation task with the goal of reaching a certain part of the world. The test problem uses the topological map in Figure 3.9, with the topological connections defined in Table 3.1. The map, based on the RoCKIn@Home scenario, was first created by the robot in the simulator using a laser, and was later marked with colours that identify the rooms. The problem definition, in terms of its state and action space, is defined in Table 3.2.

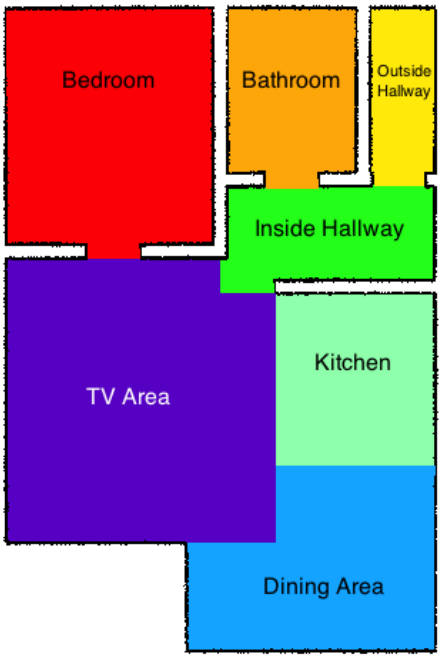


Figure 3.9: Map of the Testing Environment.

State	Action			
	Up	Down	Left	Right
IsInBedroom	-	IsInTVArea	-	-
IsInBathroom	-	IsInInsideHallway	-	-
IsInInsideHallway	IsInBathroom	IsInTVArea	-	-
IsInDiningArea	IsInKitchenArea	-	IsInTVArea	-
IsInTVArea	IsInInsideHallway	IsInDiningArea	IsInBedroom	IsInKitchenArea
IsInKitchenArea	-	IsInDiningArea	IsInTVArea	-

Table 3.1: Topological Map for the Testing Scenario

State	Name	Color
s_0	IsInBedroom	Red
s_1	IsInBathroom	Orange
s_2	IsInInsideHallway	Green
s_3	IsInDiningArea	Blue
s_4	IsInTVArea	Purple
s_5	IsInKitchenArea	Pale Green

(a) States

Action	Name
a_0	Up
a_1	Down
a_2	Left
a_3	Right

(b) Actions

Table 3.2: Problem Definition of the Testing Environment

Defining the goal of the task as reaching state *IsInBathroom*, we use the following reward model:

$$R(s,a) = \begin{array}{c} \\ s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{array} \begin{array}{c} \begin{array}{cccc} a_0 & a_1 & a_2 & a_3 \end{array} \\ \begin{pmatrix} -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ +100 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{pmatrix} \end{array}$$

which only provides a positive reward for reaching the goal state (more specifically, the positive reward is attributed to performing actions that lead to the final state, which in our scenario only happens when executing action a_0 in state s_2). A negative reward is given to every other state-action pair, punishing any decision that does not lead to the goal state.

Given the problem definition, the correct optimal policy should be:

$$\pi(s) = \begin{array}{c} \begin{array}{cccccc} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{array} \\ \begin{pmatrix} 1 & x & 0 & 2 & 0 & 2 \end{pmatrix} \end{array}$$

where "x" stands as "do not care", because the action performed at the goal state is irrelevant, and where the numbers elsewhere correspond to actions, i.e., 0 is a_0, 1 is a_1, and so forth. Independently of the algorithm used, the value-function $Q(s,a)$ is always initialized as zeros.

In every experiment, the ε parameter is set to $e^{-1/50t}$, the α parameter is set to $1/t$, the γ parameter is set to 0.9. For the algorithms with the eligibility traces extension, a value of 0.9 was used for λ, taking as much information from previous visited states as possible, without having to fully wait for the learning episode to end. Finally, throughout the experiment, the policy update frequency parameter was set to 1 and the reward for impossible actions was set to -1.

3.3.2 Results

We let the environment be explored for 500 decision episodes, and present the results in Table 3.3.

Algorithm	Resulting Q-Values	Resulting Policy
SARSA	$Q(s,a) = \begin{pmatrix} -3.868 & 416.575 & -3.649 & -3.791 \\ 0 & 0 & 0 & 0 \\ 520.484 & 415.021 & 462.368 & 465.602 \\ 77.921 & 17.082 & 414.955 & 13.739 \\ 467.346 & 69.309 & -4.468 & -5.289 \\ -4.877 & 65.775 & 415.381 & -5.197 \end{pmatrix}$	$\pi(s) = \begin{pmatrix} 1 & x & 0 & 2 & 0 & 2 \end{pmatrix}$
SARSA(λ)	$Q(s,a) = \begin{pmatrix} -92.790 & 613.028 & -83.108 & 28.271 \\ 0 & 0 & 0 & 0 \\ 828.052 & 497.808 & 529.145 & 465.187 \\ 126.135 & -31.338 & 559.346 & 14.234 \\ 766.108 & 194.726 & 58.005 & 155.240 \\ 32.655 & 77.478 & 636.426 & 126.926 \end{pmatrix}$	$\pi(s) = \begin{pmatrix} 1 & x & 0 & 2 & 0 & 2 \end{pmatrix}$
Q	$Q(s,a) = \begin{pmatrix} 320.84 & 410.020 & 335.808 & 286.032 \\ 0 & 0 & 0 & 0 \\ 518.107 & 414.223 & 458.784 & 453.183 \\ 334.147 & 332.232 & 408.934 & 330.431 \\ 464.922 & 352.164 & 363.090 & 365.797 \\ 342.720 & 326.137 & 412.516 & 292.920 \end{pmatrix}$	$\pi(s) = \begin{pmatrix} 1 & x & 0 & 2 & 0 & 2 \end{pmatrix}$
Q(λ)	$Q(s,a) = \begin{pmatrix} 270.045 & 375.869 & 220.501 & 243.215 \\ 0 & 0 & 0 & 0 \\ 479.948 & 310.829 & 308.599 & 350.306 \\ 206.605 & 226.841 & 374.179 & 279.459 \\ 434.130 & 323.702 & 326.922 & 330.658 \\ 226.913 & 222.966 & 373.843 & 201.988 \end{pmatrix}$	$\pi(s) = \begin{pmatrix} 1 & x & 0 & 2 & 0 & 2 \end{pmatrix}$

Table 3.3: Testing Results

Analyzing the results, the most important observation to be made is that every algorithm reached the correct final policy. This serves as empirical proof that the system is correctly implemented, as the obtained results are correct. Another observation that can be made is that the values for state s_1 are always zero. This is due to the fact that we stopped the learning episode after the final state was reached, meaning that being in that state is never experienced.

Figure 3.10 shows the evolution of the Q-Values for each algorithm for the $(s,a) = (2,0)$ action-state pair, which is the final pair for this scenario.

In terms of the algorithm's convergence, it can first be stated that, contrary to Q-Learning, SARSA converges to a suboptimal policy. This can be observed in the figure, since both versions of Q converge to around the same value, while both versions of SARSA converge to different values. It can then be stated that the eligibility traces versions of the algorithms converge in a noisier fashion than their TD(0) counterparts, which is due to the fact that every visit to a specific state affects the values of other states. In Q(λ), every state-action pair for the visited state is affected. Additionally, every state-action pair that led to the current pair is also affected by the visit. In SARSA(λ), the latter also occurs.

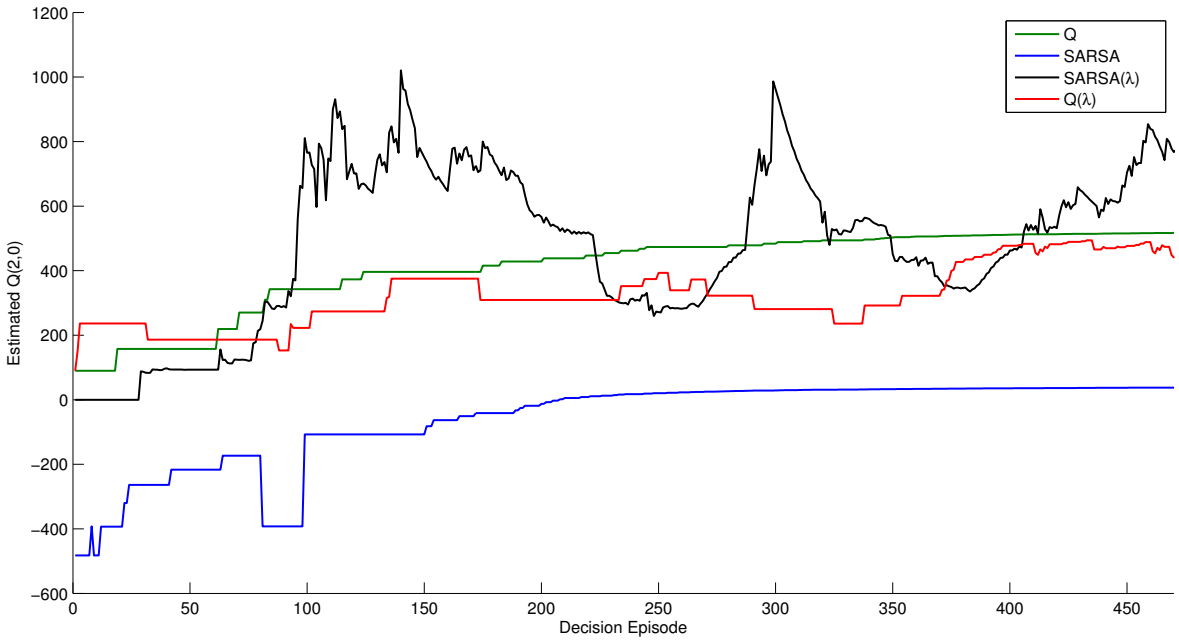


Figure 3.10: Convergence of the Test Results.

The obtained results are as expected. Both Q and SARSA converge in a noiseless fashion, while SARSA converges to a suboptimal value. Q(λ) converges to the optimal value, while doing so noisily, and SARSA(λ) appears to oscillate around the optimal value in a noisy fashion.

Assuming that the value that Q converges to is the optimal one, these results show that Q is the algorithm that converges faster. However, in the general case, Q(λ) is theoretically excepted to converge faster than Q.

To conclude on the testing results, we believe that, due to their apparent correctness, they provide empiric proof of the correct implementation of the system.

Chapter 4

Task Plan Representation for the SocRob@Home Robot

Defining and developing a task-level software structure for the SocRob@Home robot is another goal of this thesis. MDM, serving the purpose of being a solution for task-level definition, could be used exclusively to accomplish that. However, there are simpler methods for defining deterministic tasks, where uncertainty is not a factor. The discrete system's Finite State Machine (FSM) framework serves that purpose, and given that it is both simpler to design than DT-based methods and that there are already several off-the-shelf solutions for it, we have decided to use it in our situation. ROS already has a solution for the development of FSMs, which is the SMACH[1] package. This package provides a well documented and simple to implement API, and therefore we will use it to handle the deterministic segments of tasks.

In this chapter, we start by explaining the FSM that was developed for the Robótica 2014[2] competition, where the SocRob@Home team participated. We then identify the fact that the fully deterministic system has several limitations, namely in the sense that it does not cope with uncertainty. Given this, we propose using MDM along with SMACH to be able to overcome those limitations by introducing decision-making under uncertainty. Additionally, the RL capabilities allow overcoming yet another issue, which is that of the complexity of the environments, by allowing the robot to learn the optimal behaviour instead of having to model the environment, which in many circumstances is not feasible. Finally, we give a description of the simulator that was created to facilitate the learning process.

4.1 Finite State Machines using SMACH

SMACH is a package that maintains an API for designing FSMs in the ROS middleware. For this purpose, it provides several different state implementations that differ regarding their outcomes and on

[1] `http://wiki.ros.org/smach`, as of September 2014

[2] `http://www.robotica2014.espe.pt/index.php/en/`, as of September 2014

how they function. It also provides an architecture basis in which every state outcome connects to a different state. These two characteristics together allow for designing FSMs which, when allied with SMACH's easy-to-use API, provide a good basis for deterministic task definition.

Before continuing, it is important to make the distinction between a state as it is defined in the DT framework and in the Discrete Event Systems framework. In the former, a state is a unique representation of the world, being defined over a unique collection of perceptions that are different from those of every other state. In the latter, however, a state has a broader meaning.

(Cassandras and Lafortune, 2008) defines the state of a system as a description of its behaviour at that time in some measurable way. More formally, it defines a state at t_0 as the information required at t_0 such that the system's output $y(t)$, for all $t > t_0$, is uniquely determined from this information and from its input $u(t)$, $t > t_0$.

In the context of task-level definition, a state, when compared to the DT terminology, since it is a description of the system's behaviour, can either be a state, an behaviour or an arrangement of both. Therefore, states are the only components of FSMs, and they are connected among themselves by outcomes, which describe the transitions between them. Each state can be reached by many other states and can also lead to many other states, provided that there are as many outcomes connecting them. In the graphical representation of FSMs that is used in this thesis, states are portrayed as ovals with the state's name in uppercase letters and outcomes are portrayed as arrows with the outcome's name in lowercase letters. Figure 4.1 contains an example FSM that represents the task of a barkeeper robot.

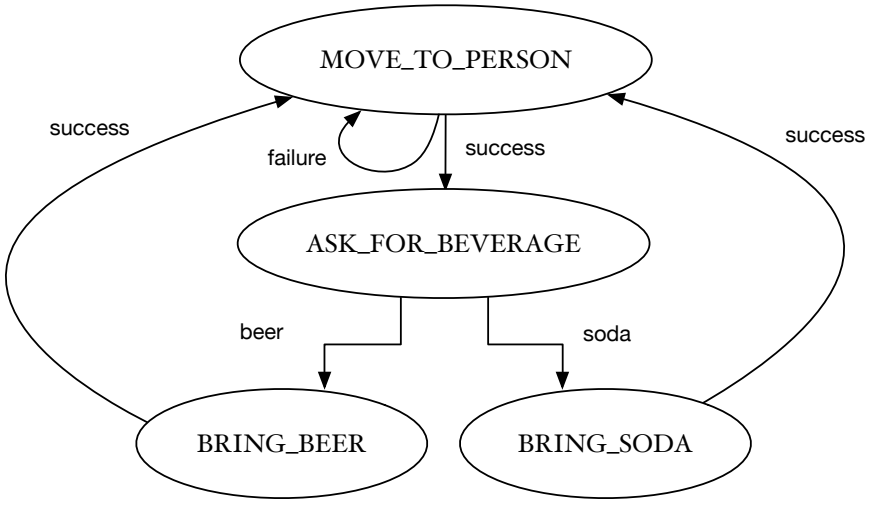


Figure 4.1: Example FSM. The states are *MOVE_TO_PERSON*, *ASK_FOR_BEVERAGE*, *BRING_BEER* and *BRING_SODA*. The initial state is *MOVE_TO_PERSON*. If, for instance, in the *ASK_FOR_BEVERAGE* state, the robot identifies that the person asked for a beer, then a transition to *BRING_BEER* is executed.

The aforementioned different state implementations provided by SMACH are the following:

- **State**: no pre-defined outcomes or features;

- **SPAState**: three pre-defined outcomes - succeeded, preempted and aborted. No pre-defined features;
- **MonitorState**: three pre-defined outcomes - valid, invalid and preempted. The pre-defined feature of subscribing to a ROS topic, defining its outcome based on the information received through it;
- **ConditionState**: two pre-defined outcomes - true and false. Has the feature of checking a condition and defining an outcome based on it;
- **SimpleActionState**: the same pre-defined outcomes of the SPAState, but with the feature of being a proxy to a simple actionlib[3] action.

Another functionality provided by SMACH is abstract containers. A container, among other structures, can be a state, an FSM or a concurrence node (which is a node that allows for parallelism between states, that can either end when all of the states finish executing, or can end when one of the states does so, preempting the execution of all of the other states). Containers can also incorporate other containers, which allows the creation of complex FSMs. For instance, imagine that the topmost container is an FSM, which is naturally composed of states. However, since states are also containers, one of them can be a concurrence node, which in turn also contains states. One of the states of that concurrence node, also being a container, can, for instance, be an FSM instead of a state.

As can be seen, then, SMACH grants the possibility of creating complex FSMs, which, allied with its easy-to-use API led us to choose it as our tool for deterministic task design.

4.1.1 Task for Robótica 2014

During the development of this thesis, the SocRob@Home team participated in the Robótica 2014 competition, specifically in the Freebots league, which allows every participant team to decide by itself which task or functionality to show off. Since the goal of the team is to participate in the @Home competition, it was decided to prepare a task related to an @Home setting. The created task was fully deterministic and was implemented using SMACH. It served as a first test for using this method for implementing the deterministic segments of tasks in the SocRob@Home robot.

The SocRob@Home team ended up being the only participant in the Freebots segment, however, the demos went as expected and the performance was competent, with several public demonstrations attracting many visitors. In addition, some additional unplanned behaviours ended up being implemented

[3] Actionlib is another ROS package that has a client-server structure where the client asks for the server to execute goals. The package's usage with SMACH is based on having the client implemented in an FSM while having the executor server in another ROS node, thus allowing for executing actions outside of the FSM environment. `http://wiki.ros.org/actionlib`, as of September 2014

(mainly interactions between the robot and an automated home, e.g. opening a door through the network) and integrated into the FSM during the competition, which serves to state SMACH's flexibility as well as ease of use. Although we do not explain it in depth, the final version of the FSM, which uses several of SMACH's features, such as concurrent states, can be seen in Appendix A.

4.1.2 Integration of MDM with Finite State Machines

It was mentioned before that the idea behind adding RL to the MDM package is to provide the user with the possibility of using it to have the agent learn the optimal policy for a certain MDP, later using the implemented DT methods to run the MDP with that policy. Using SMACH along with MDM further extends its reachability, since it makes it possible to combine (PO)MDPs with FSMs, covering both deterministic and stochastic environments.

One could design an autonomous task by first defining the world states and problem representation, later identifying where uncertainty is present and where learning makes sense or is an advantage. In this situation, the segments of the problem in which there is no uncertainty can be represented by deterministic transitions between states as defined in the FSM context. The segments where uncertainty is present (pending an observability analysis) can be defined by (PO)MDPs. In those, if a model of the environment is too complex to determine, RL can be used to compute the optimal policy.

There are two ways to integrate SMACH with MDM. The first one is to have an FSM state be a (PO)MDP. This is possible due to the fact that MDM controllers can be started and stopped through a ROS service. The other way is to have an action of the (PO)MDP launch an FSM.

Figure 4.2 shows a simple example of the integration of SMACH with MDM. Suppose that the only non-deterministic segment of this task is represented by the *TAKE_OBJECT_TO_PERSON* state, and also that the subtask for that state is implemented as a (PO)MDP created with MDM. When that state is reached, the FSM can start the controller for the (PO)MDP, wait for it to finish its execution, and stop it. Suppose now that there is an action in that (PO)MDP that is a complex behaviour, possibly being better represented by a simple FSM. The definition of that task in the Action Layer can be made, in a similar way, to start and stop a SMACH controller, allowing for creating an FSM to execute the action.

We propose that this integration between SMACH and MDM sets the basis for the task-level definition of the SocRob@Home robot. This methodology will be followed throughout the rest of the thesis.

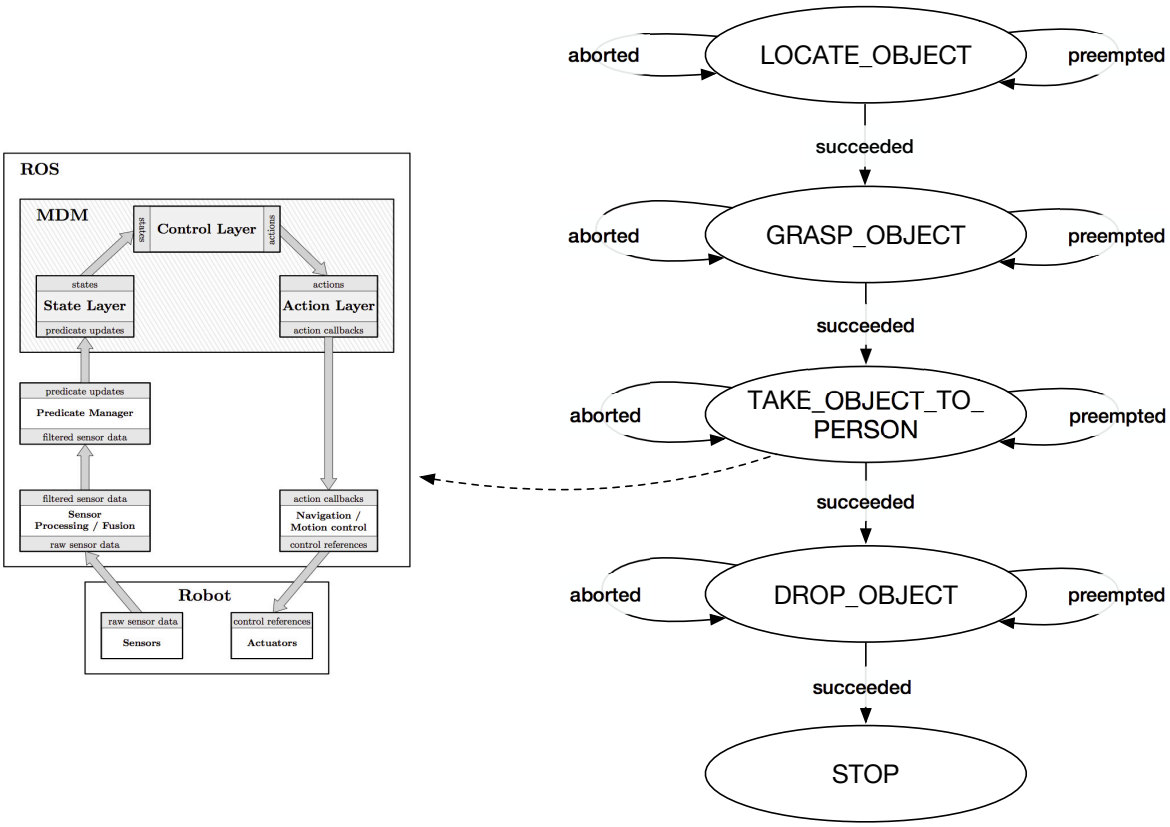


Figure 4.2: MDM Integrated within SMACH.

4.2 The Simulator

A simulator is an important tool to have when developing a robot. It provides a platform for testing individual behaviours, navigation, perception, task integration and decision-making, among others. Additionally, a simulator is useful for learning systems, due to the fact that it provides a platform where the robot can experience its environment without actually having to deploy the real robot, which in most cases would be slow due to concerns such as limited battery duration, real time execution (in general, simulators are able to simulate in faster speeds), etc. As such, and since there was no simulator for the SocRob@Home robot, we proposed to develop one.

ROS has two simulators built within it: Stage[4] and Gazebo[5]. Both of them provide a physics engine to compute the dynamics of both the robot and the world surrounding it. Stage simulates in 2D while Gazebo simulates in 3D. Due to the fact that manipulation is one of the targets for future simulations, 3D simulation is required. As such, the simulator that we chose was Gazebo.

[4] `http://wiki.ros.org/stage`, as of September 2014
[5] `http://wiki.ros.org/gazebo`, as of September 2014

4.2.1 A Description of Gazebo

There are two separate components to a Gazebo simulation: the world description and the robot model. The world description is built by including models, which can either be static or dynamic, that compose the world. There are several pre-built models available in Gazebo, but creating one in a 3D modelling software and building one through an assembly of geometric shapes is also possible. The robot model is built by defining links, which contain meshes and a physical description, and by defining joints over pairs of links where movement is possible.

At run-time, Gazebo published information to other ROS nodes using topics and services. That information can then be utilized to perceive the world.

4.2.2 Building the World Model

Given the team's future participation in the RoCKIn competition, specifically in the @Home segment, we created the world based on the blueprint of the home that will be used in it, which is shown in Figure 4.3.

The walls were defined as static models based on the box geometric form. The patio walls were defined to be made of glass, which does not reflect lasers. A static table was added as sample furniture.

This simple version of the world is enough to fulfill the simulation needs for this thesis (which are mainly navigation). Dynamic objects will eventually be needed to simulate manipulation. However, their inclusion is fairly straightforward, since the only requirement is to provide a simple description of the object in the world description file.

4.2.3 Building the Robot Model

The robot model was created based on the team's current robot, which can be seen in Figure 4.4. Some of the meshes used already existed from the model of the SocRob Soccer robots, and the ones that had to be created were developed by other members of the team. Geometric descriptions of the links were measured from the robot itself and the dynamic joints were created according to the real robot as well.

Since Gazebo only has differential drive and skid drive controllers implemented off-the-shelf, and since the robot is holonomic, a controller had to be developed for the robot's motors. This process required several steps. First, three transmissions, which are components built into Gazebo, had to be implemented on top of each wheel joint. Then, three controllers, which are also components build into Gazebo, had to be created for each transmission. The controllers are simple PIDs and the control variable is the velocity of each wheel. To simulate the omnidirectional wheels, we associated null friction to movement perpendicular to the wheels' primary axis. Finally, a ROS node also had to be built to translate the robot's navigation module's control messages to messages understood by Gazebo.

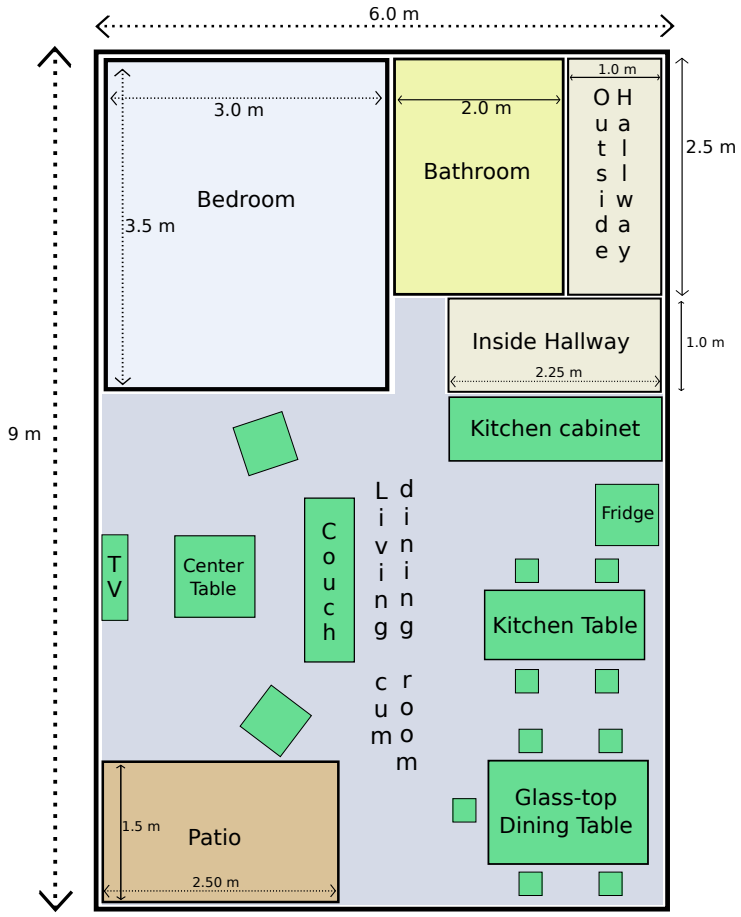


Figure 4.3: Testbed Blueprint. Reprinted from (Schneider et al., 2014).

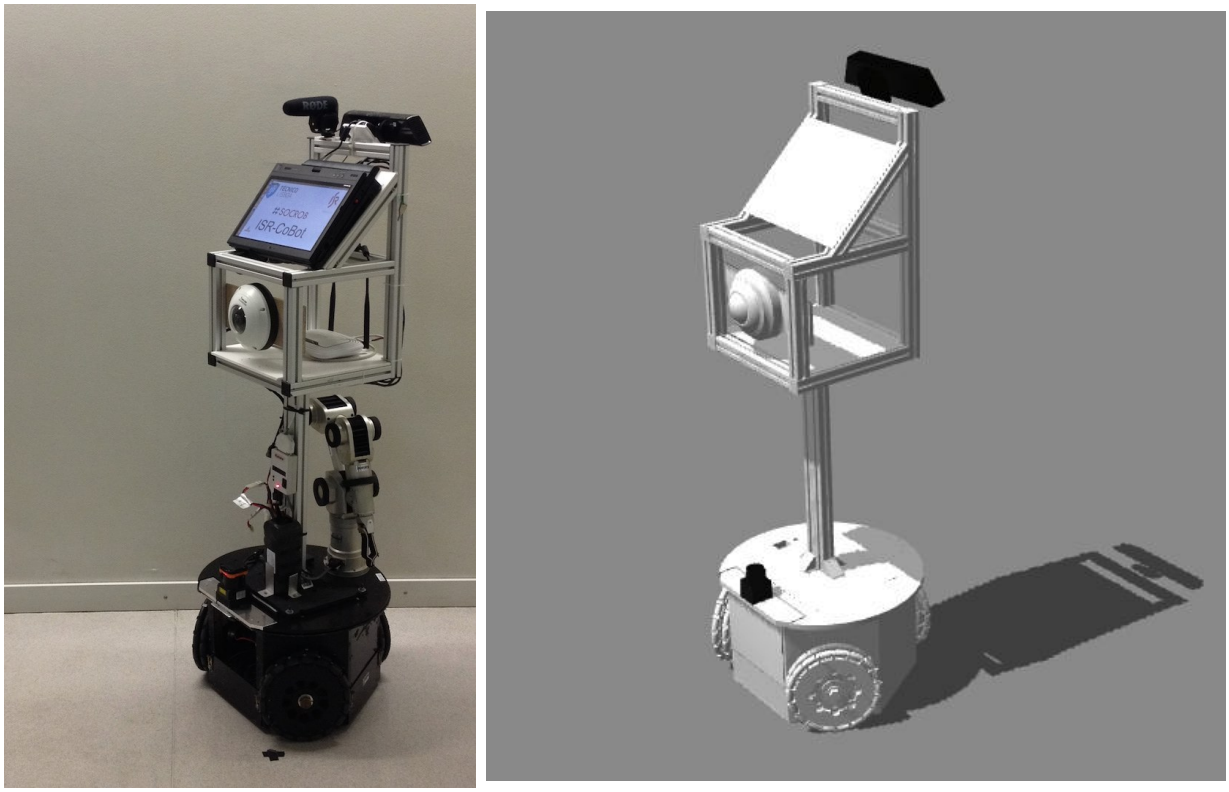
There are some differences between the real robot and the model that was created. As was mentioned before, manipulation is not a subject of this thesis, and as such we did not include the arm in the model. The wireless router, the microphone and the touchscreen were also not included, given that they do not serve any purpose in simulation. Besides these, the cameras were included in the model but do not function in simulation, and the laser is included in the model and does function in simulation.

4.2.4 Integration with the Robot's Software

At the moment, the simulator only handles navigation. The laser and motor encoders provide information so that Gazebo can compute and output odometry information, which is then handled by the robot's navigation code as if it were coming from the real robot.

The arm can later be added by appending its model to the robot's and creating controllers for each arm joint. The cameras can also be simulated by modifying how they are defined in Gazebo. These two additions can allow integration with the closed-loop manipulation code in the future.

Figure 4.5 shows the final result, with the robot and the created world within the simulator.



(a) The SocRob@Home Robot.

(b) The Simulated SocRob@Home Robot.

Figure 4.4: The Robot.

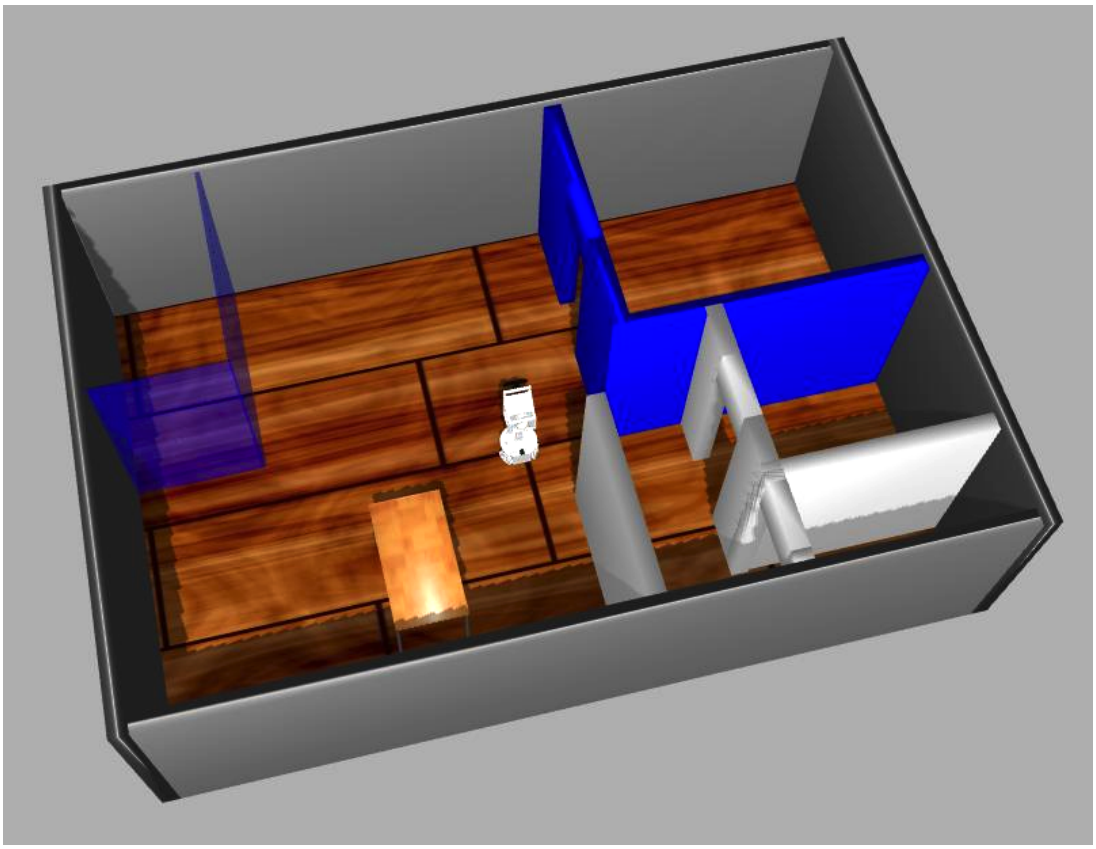


Figure 4.5: The Simulator.

Chapter 5

A Case Study in Reinforcement Learning Applied to Domestic Robots

To provide an example application of our proposed model for task definition of the SocRob@Home robot, we created a Case Study, which is based on one of the tasks that the robot will have to perform in the RoCKIn Competition 2014. Appendix A contains pseudo-FSMs of the RoCKIn tasks that were designed by the team. From them, we identified that the task in Appendix B.1 contains a subtask subject to uncertainty. That subtask is the focus of this Case Study.

This chapter explains our rationale while modelling the subtask, dividing it into deterministic and stochastic segments. It also explains how we defined the stochastic segments as MDPs, and how we used RL techniques to cope with the complexity of the problem. Finally, the results are presented and compared to a naive approach to solving the problem.

A repository containing the complete output of all of the results presented in this chapter is available in `https://github.com/ptresende/thesis.git`.

5.1 Identifying the Task and Designing its Structure

Figure 5.1 contains the subtask of the FSM in Appendix B.1 that we identified to be subject to uncertainty. The subtask begins with the robot going to the homeowner, later receiving a command, looking for and bringing back a certain object. There is also the possibility that the robot has prior knowledge about the object's location. When there is no prior knowledge, or when that knowledge is incorrect, there is uncertainty in the object's location.

For the subject of our problem, we consider the dialogue in the FSM's second state as following a deterministic set of rules. Therefore, the first two states are fully deterministic. The search state with prior knowledge is also deterministic, unless the knowledge is incorrect, in which case we consider the

same solution as we do for the state with no prior knowledge.

With this, we designed the FSM in Figure 5.2 to represent the problem. The first state takes care of initializing the manipulator. The second state is a simple navigation to a preset location where the person is, which corresponds to the first state of the pseudo-FSM. The third state is the dialogue state that also corresponds directly to the pseudo-FSM's second state. We then divide the problem into two subproblems: the case with prior knowledge and the case without it.

The first state of the prior knowledge segment is a navigation to the location where the object is supposedly known to be. If the object is found there, the next state is to move to the person's location and to give him/her the object. If, however, the object is not found, the next state is a sub-FSM representing a naive search all over behaviour to search for it (the behaviour consists of going to every room and looking for the object; due to the size of the sub-FSM, we do not present it here). In the segment without prior knowledge, the search MDP is the first and only state (the MDP will be explained in the next section).

If, following the MDP's policy, the robot is successful in finding the object, the task is complete. Otherwise, the next state is the search all over behaviour.

We base the environment for this Case Study on that of the RoCKIn competition, which was already used when testing the implemented system and is shown in Figure 3.9, with the topological connections defined in Table 3.1.

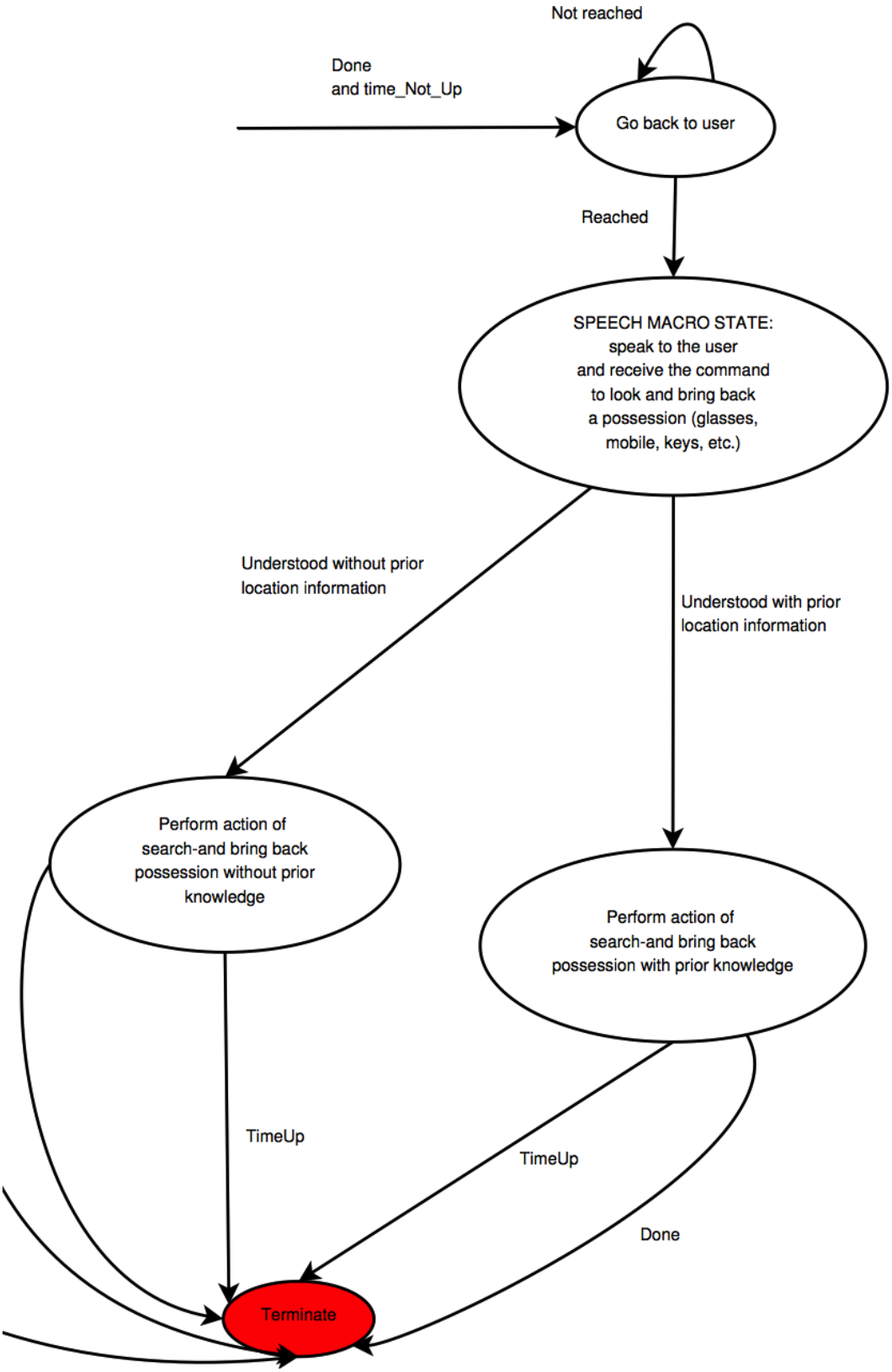


Figure 5.1: The Subtask.

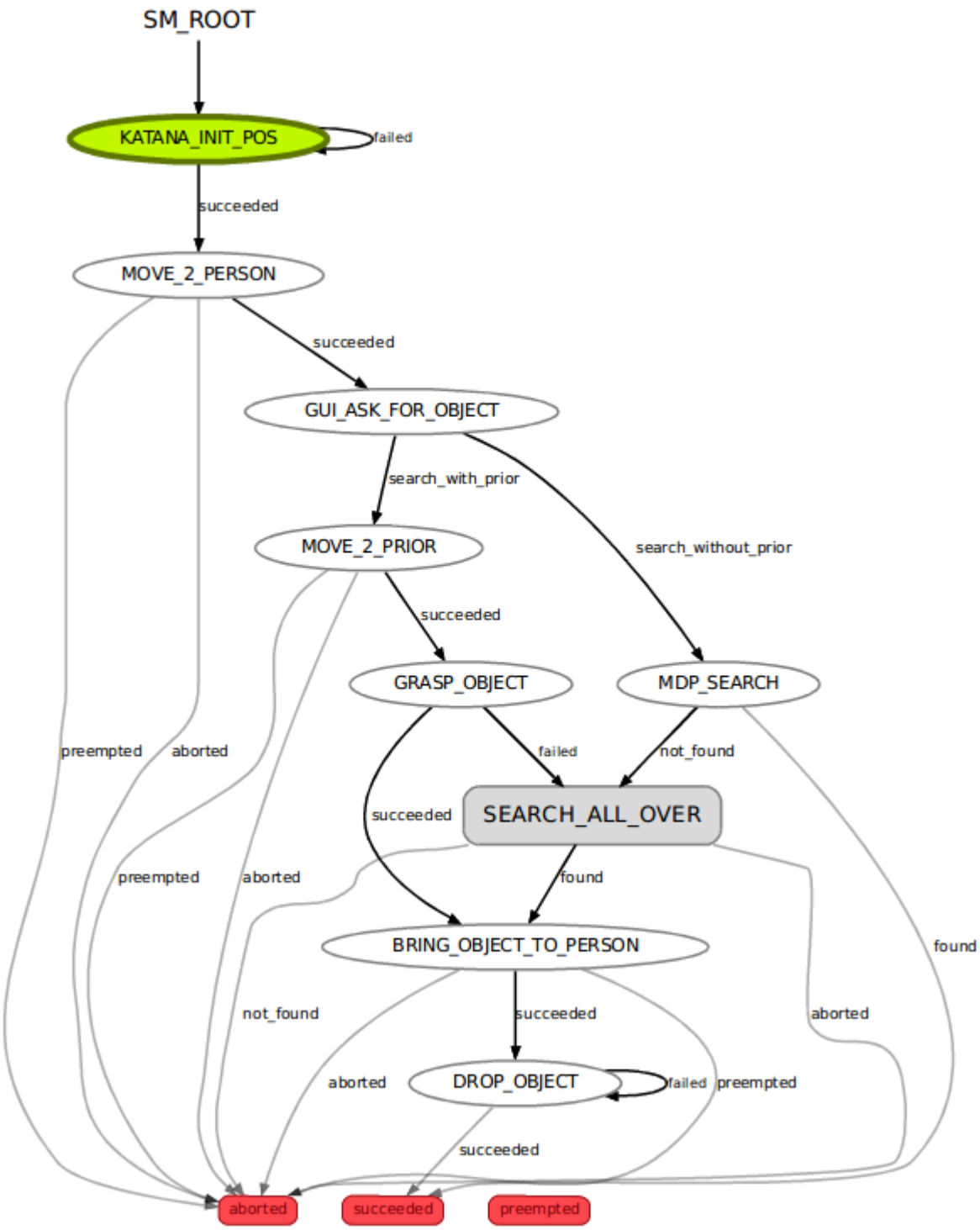


Figure 5.2: The FSM for the Case Study.

5.2 Learning to Find an Object

The subtask to be represented by the MDP is the task of searching for and bringing back the object to the person. For instance, while a certain person might usually keep their book in the bedroom, another one could usually keep it in the living room. As such, given the different specific application scenarios of this task, it is fitting to have the robot learn how to best perform it accordingly to each specific scenario. To accomplish this, we use the RL techniques that we implemented in MDM to obtain the optimal policy by experience. In this section we go over this process, explaining the problem definition and our application. Optimally, performing this task should require the least amount of decision episodes to locate the object, later grasping it, once again taking the least amount of decision episodes to get to the person, finally releasing the object.

5.2.1 Problem Definition

We mentioned in Subsection 3.1.1 that, in MDM, the states are defined based on predicates which are interpreted into state factors. In this case, we decided to define our states over four state factors, representing the robot's location, whether the object is found, whether the object is in the robot's possession and whether the person is found. Each of these state factors has several different possible values:

- **Robot Location ($\mathcal{X}_1$):** *IsInBedroom*, *IsInBathroom*, *IsInInsideHallway*, *IsInDiningArea*, *IsInTVArea* and *IsInKitchenArea*;
- **Object Possession ($\mathcal{X}_2$):** *IsObjectPossessed*;
- **Object Found ($\mathcal{X}_3$):** *IsObjectNotFound*, *IsObjectFoundWithLowConfidence* and *IsObjectFoundWithHighConfidence*;
- **Person Found ($\mathcal{X}_4$):** *IsPersonFound*.

The "object found" state factor could be defined as a simple predicate representing whether the object is found by the robot. However, this predicate would be partially observable. This is due to the fact that the detection process is inherently noisy, and as such the state of the object is actually partially observable. Since RL theory applied to POMDPs is still an open subject, as we mentioned in Subsection 3.2.8, we circumvent this issue by defining three confidence levels for having found the object, where *IsObjectNotFound* can be considered as the lowest (meaning that almost certainly the object was not found). If the "person found" state factor was dependent on image recognition algorithms, the partial observability issue would also arise. However, in this situation, since the person is assumed to stay in the same room in which the request to find the object was made, its value is only dependent on whether the robot is in the room where the person is known to be.

With this predicate definition, we define a state as $s =<$ Robot Location, Object Possession, Object Found, Person Found $>$. Given the number of possible values for each state factor, the size of the state space is $|\mathcal{S}| = 6 \times 2 \times 3 \times 2 = 72$.

Having defined the state space, we now need to define the action space. Given that part of the problem is navigation, we define four movement actions: Up, Down, Left and Right. These actions are the same that were used in Section 3.3. They are created using the Topological Tools package that is included in MDM and serve the purpose of navigating a topological map. Besides the navigation actions, we define two additional actions: *Grasp Object* and *Release Object*. These actions command the robot to use its arm to pickup and drop the object, respectively. The size of the action space is $|\mathcal{A}| = 6$.

5.2.2 Reward Strategy

Reiterating, the end goal of the MDP task is to bring back the desired object to the person. To do so, we reward positively performing the *Release Object* action when in a state that satisfies the following conditions: the object is in the robot's possession and the person is found. The other predicates' values are irrelevant to the state where performing the action should be rewarded. Translating to the predicates definition, this means that both *IsObjectGrasped* and *IsPersonFound* must be true.

Given the size of the reward matrix, we do not include it here, however, a $+100$ reward is attributed to the state-action pairs where the aforementioned predicates are true, while a -1 reward is attributed otherwise. These numbers follow the same strategy that we used while testing the software in Section 3.3, where the negative reward serves as punishment for every decision taken that does not lead to the goal state. We also attribute a -1 reward for impossible actions, punishing them only as much as taking a bad decision. The only positive reward that is possible to be received is when the goal state is reached, which guarantees that the value of every state-action pair that eventually leads to it will be positively reflected by the reward. This means that the policy will eventually converge to optimally perform the task as we define it.

5.2.3 The Parameters

A summary of the parameter methodology that we used is shown in Table 5.1.

The discount rate γ parameter determines how much weight is put into future rewards versus how much is put into more immediate rewards. Given that we only award a positive reward on the last decision before the end of the task, it seems logical that it is beneficial to put as much weight as possible into future rewards. That way, we guarantee that every decision takes into account the possibility of receiving the highest reward, instead of having updates that only take into account the intermediate punishing rewards. As such, we set $\gamma = 0.9$.

The ε parameter, on the other hand, plays a very noticeable role in the exploration versus exploitation problem and can influence the learning speed distinctly. In the testing scenario from Section 3.3, we had a situation where learning could be done "continuously" due to the fact that there was no real final state. In the Case Study, however, there is a final state and therefore learning has to be done in runs. We define a run in this situation as the process that begins with the robot having been asked to find the object, and ends with it releasing the object nearby the person. Using runs, then, the ε value cannot be updated continuously as it was in the testing scenario. We therefore decided to perform each run with a constant ε value and to update its value in-between runs. This led to the need of experimenting with how many runs should be done with which value. We concluded empirically that a good ε progression for this scenario is performing 20 runs with $\varepsilon = 0.9$, 20 runs with $\varepsilon = 0.7$, 60 runs with $\varepsilon = 0.5$ and as many runs as necessary for convergence with $\varepsilon = 0.3$.

Regarding the learning rate α parameter, it influences directly how much weight is put into each Q-Values update, functioning as a gain for the error correction element of the update. Since, in our scenario, the environment is static, we decided to decrease α with time (which guarantees convergence of the Q-Values update rule for static environments). We conducted an empirical study similar to what with did for ε and concluded that the same progression provides good results.

Finally, regarding λ, since we want learning to be as fast as possible, we set $\lambda = 0.9$. This guarantees that, while still not turning our TD(λ) algorithm into an MC method, we make the most of using eligibility traces to accelerate learning as much as possible.

# Runs	ε	α	γ	λ
20	0.9	0.9	0.9	0.9
20	0.7	0.7		
60	0.5	0.5		
-	0.3	0.3		

Table 5.1: Parameters for the Case Study

5.2.4 Learning Method

In terms of applicability, the greatest difference between SARSA and Q-Learning is that the latter learns the optimal policy independently of the exploration, while the former takes the exploration directly into account at every backup. The effects of this difference are explored in Example 6.6 of (Sutton and Barto, 1998), where the authors present a situation where the agent must learn a path along a cliff. At every transition, a negative reward of -1 is given, except when the agent falls into the cliff, where a negative reward of -100 is given. Q-Learning learns the optimal path, i.e., the path closest to the cliff. SARSA, on the other hand, learns a non-optimal path farther away from the cliff. The accumulated reward per learning episode is higher while using SARSA than Q-Learning. This is due to the fact the

ε-greedy action selection method leads the Q-Learning agent to sometimes fall off the cliff, while the same situation does not happen with SARSA since it travels farther away from the cliff.

While this is not directly related to our implementation of the Case Study, it is a relatively common scenario in robotics applications. In fact, if we had dealt with impossible actions with the state-conditional action spaces methodology (see Subsection 3.2.4), we could have defined a -100 reward for performing the *Release Object* action in any situation where the robot was not in the same room as the person. In this scenario, the equivalent of falling off the cliff would be, for instance, releasing a glass of water away from the person, causing it to drop on the ground and break. As such, to increase the Case Study's generality, we decided to use SARSA.

Regarding the decision of whether or not to use eligibility traces, we decided to use them in every scenario given that they present a noticeable improvement in learning speed. Also, despite having decided to use SARSA as our learning method, we also present the results of a Q-Learning run.

5.2.5 Simulating the Environment

There are two different components that need to be simulated in the Case Study scenario. The first is robot and requires a navigation simulator. The second is the person and the object, which require managing their positions and reactions to the robot's actions.

We decided not to use the simulator developed in Section 4.2 due to the fact that it was too heavy in terms of processing requirements (due to simulating in 3D) to be reasonable to perform several simulations on. We therefore used the aforementioned Stage simulator, which works in 2D, and which allowed us to run simulations at almost 7 times faster than real time, whereas with Gazebo we were not able to simulate past real time speed. This does not reduce the quality of simulation due to the fact that Stage's physics simulator is as realistic as Gazebo's for planar robots.

Regarding the second component of simulation, we created a simple ROS node that is able to communicate with the rest of the MDM package, as well as with the Stage simulator. The goal of this component is to provide information to the Predicate Manager package, allowing it to infer on the current value of the three predicates *Object Possession*, *Object Found* and *Person Found*. This simulator contains within it the MDP transition model, which, if learning was to be performed online, would be encompassed in the real environment.

When the simulator is first initialized, the person's position is attributed to the Dining Area and the object position is attributed either to the Bedroom with a 60% chance or to the TV Area with a 40% chance.

The simulator operates based on either receiving a new position of the robot or on receiving either a *Grasp Object* or *Release Object* action. When a new position is received, the value of the predicates *Ob-*

ject Found and *Person Found* is updated accordingly. The former predicate value is attributed based on whether the robot is in the same room as the object. If it is not, the confidence level becomes *IsObjectNotFound*. If it is, the confidence level has a 50% chance of becoming *IsObjectFoundWithLowConfidence* and a 50% change of becoming *IsObjectFoundWithHighConfidence*. The latter predicate value becomes true if the robot's position is equal to the person's position and false otherwise. Upon receiving either a *Grasp Object* or *Release Object* action, the simulator updates the value of *Object Possession* accordingly, in a deterministic manner.

To deal with the impossible actions in this scenario, i.e., trying to perform a *Grasp Object* action when the object is already grasped or a *Release Object* action when the object has not yet been grasped, we used the functionality created in Subsection 3.2.4.

5.2.6 Applying MDM to the Case Study Scenario

To apply MDM to the Case Study scenario, we had to implement instances of the Predicate Manager, the State Layer, the Action Layer, and the Learning Layer.

In the instance of the Predicate Manager, we defined the predicates and their possible different values. We also defined how they communicate with the simulator/robot's sensors (when applying the Case Study to the real robot) to update their value. In the State Layer, we defined how the predicates are mapped into state factors. In the Action Layer, we defined the actions and how they communicate either with the simulator or with the real robot. Lastly, in the Learning Layer, we instantiated our learning algorithm, along with its parameters.

When applying the Case Study to the real robot, since we had already obtained the policy from learning offline, instead of instantiating a Learning Layer, we instantiated a Control Layer that implemented the learned policy. Besides this simple alteration to the code, we only had to modify the Predicate Manager and Action Layer to communicate with the real robot instead of with the simulator.

5.3 Results

5.3.1 The Learned Task

Appendix D contains the output of both a SARSA(λ) and a Q(λ) learning task in the form of the final Q-Values as well as the final policy for the Case Study scenario.

The resulting policy for the Case Study scenario leads to the execution represented in Table 5.2.

To contrast the results, we created another scenario where the object is always in the Bathroom and where the person is always in the Bedroom. The resulting policy for this contrasting scenario leads to the execution represented in Table 5.3.

State				Action
Robot Location	Object Possession	Object Found	Person Found	
IsInTVArea	False	0	False	Left
IsInBedroom	False	1 or 2	False	Grasp Object
IsInBedroom	True	1 or 2	False	Down
IsInTVArea	True	0	False	Down
IsInDiningArea	True	0	True	Release Object
IsInDiningArea	False	0	True	-

(a) Object in Bedroom

State				Action
Robot Location	Object Possession	Object Found	Person Found	
IsInTVArea	False	1 or 2	False	Grasp Object
IsInTVArea	True	1 or 2	False	Down
IsInDiningArea	True	0	True	Release Object
IsInDiningArea	False	0	False	-

(b) Object in TV Area

Table 5.2: Resulting Task Execution for the Case Study Scenario

State				Action
Robot Location	Object Possession	Object Found	Person Found	
IsInTVArea	False	0	False	Up
IsInInsideHallway	False	0	False	Up
IsInBathroom	False	1 or 2	False	Grasp Object
IsInBathroom	True	1 or 2	False	Down
IsInInsideHallway	True	0	False	Down
IsInTVArea	True	0	False	Left
IsInBedroom	True	0	True	Release Object
IsInBedroom	False	0	True	-

Table 5.3: Resulting Task Execution for the Contrasting Scenario

Observing the results, we can conclude that the Case Study task is learned correctly. We can also conclude that different environment dynamics within the same task are also learned correctly. These conclusions confirm that what we defined as the purpose of our Case Study - to develop a learning system for a robot so that it can learn by itself to bring back to the homeowner a certain object - was accomplished.

5.3.2 Convergence Analysis

To prove that learning was being accomplished correctly, we studied the method's convergence. To do this, we let SARSA(λ) run in the Case Study scenario until 5000 decision episodes happened. With the

results, we computed the L^1 norm of the difference between two consecutive runs' Q-Values:

$$D(n) = \|Q^n - Q^{n-1}\|_1 = \sum_{\mathcal{S}} \sum_{\mathcal{A}} |Q(s,a)^n - Q(s,a)^{n-1}|$$

where m is the number of state-action pairs. We then decided to fit the data, using a Non-Linear Least Squares method (Marquardt, 1963), to an exponential curve (given that these methods usually converge exponentially):

$$F(x) = ae^{bx}$$

with results shown in Figure 5.3. The obtained coefficients were: $a = 43.28$ and $b = -0.01548$. Computing the limit,

$$\lim_{x \to +\infty} F(x) = \lim_{x \to +\infty} 43.28e^{-0.01548x} = 0$$

we reach the conclusion that it indicates that the method's results are converging to a difference of zero between to consecutive runs' Q-Values, meaning that it is approaching the optimal values and therefore that the method is converging.

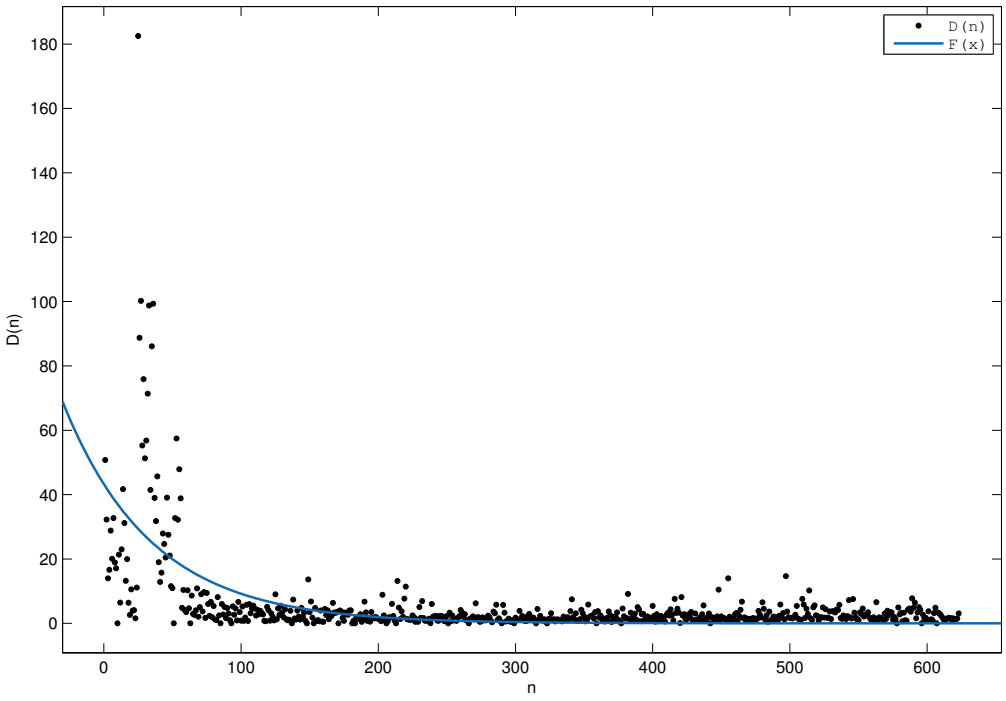


Figure 5.3: Convergence Analysis Fitting.

5.3.3 Analysis of the Effects of Maintaining the Eligibility Traces Values Between Learning Runs

While experimenting with the parameters, and given that we decided to divide the learning process in runs, we reached the question of whether or not to maintain the eligibility traces values between runs, i.e., whether to use the final values of the eligibility traces of the last run as their initial values for the next one. Both situations are possibly correct. The problem definition does not change between runs, and the environment is static, which means that the state-action pairs which were eligible in the previous run should also be in the next one, and therefore, maintaining the values makes sense and should accelerate learning. On the other hand, the exploration is different between runs, and given that the eligibility traces values impact directly the Q-Values updates, and that the updates are performed for every state-action pair in each backup, maintaining the values may lead to incorrect updates on some state-action pairs which have not yet been explored in the new run. Another possibility is that, given that the eligibility traces for unvisited state-action pairs decay relatively fast, maintaining their values between learning runs only affects the beginning of the future runs, thus not having a significant impact on the learning speed.

Given this situation, and the fact that we were not able to find any information on this topic, we decided to do an empirical analysis of the effects of maintaining the eligibility traces between learning runs by studying the difference in learning speed (which, in turn, should the values of the algorithm's parameters be kept unchanged throughout the experiment, is a measure of how maintaining the eligibility traces affects the algorithm's convergence). We performed several learning tasks that we considered to be accomplished when the correct policy was obtained. To speed up the process, we consider a correct policy as one that leads the robot from the Case Study's initial state to its final state. Whether the policy is completely optimal from every initial state is not confirmed.

Each of these learning tasks is composed of several learning runs that amount to a certain number of decision episodes. Our criteria for measuring the learning speed is the total number of decision episodes taken to reach the correct policy. We then analyze the results, by applying a one-way analysis of variance, (Box et al., 1954), method based on the two classes of results: maintaining the values and re-initializing them as zeros. The method used is SARSA(λ) and the parameters used follow the description in Subsection 5.2.3. Table 5.4 contains the number of decision episodes of several simulations that were performed for this study and that compose our data for this study.

Applying the method to the obtained results, we reach a *p-value* of 0.4084. Figure 5.4 shows a plot of the method's results. In it, the red line represents the mean and the blue boxes represent the *F-statistic*, a ratio of the mean squares. The *p-value* is interpreted as the null hypotheses that all samples provided are drawn from populations with the same mean. Values close to zero suggest that the samples are significantly different in terms of their populations means. Given the high *p-value* that we obtained, we

state that, at least in this Case Study's specific scenario, there is not a significant difference in maintaining the eligibility traces values across learning tasks versus resetting them. However, given the generic application of RL methods that we utilize, we suggest that this result may carry over to other similar experiments.

Learning Task	Resetting (# Dec. Eps.)	Maintaining (# Dec. Eps.)
1	2677	1278
2	2210	1814
3	2208	1985
4	1937	2384
5	2248	2783
6	3339	1924
7	1789	1754
8	1724	1437
9	2100	2117
10	1371	2396
11	1827	2236
12	1340	1584
13	2472	2788
14	2565	2267
15	1191	1549
16	1168	1270
17	2524	1474
18	2296	1976
19	1783	1764
20	2542	1861
Mean	2065.6	1932.0

Table 5.4: Results of Several Simulations for the Eligibility Traces Study

5.3.4 Application to the Real Robot

The robot that we planned to apply the Case Study to (which is the one that was used in the simulator described in Section 4.2), suffered a hardware malfunction during the writing of this thesis, and therefore we were not able to use it. Instead, we used the MBOT robot from the MOnarCH project[1]. This change introduced some differences in the final result, given that the MBOT does not have a manipulator, and as such we had to replace the grasping and releasing actions with dummy behaviours. Another difference is that the MBOT does not have a marker recognizer, which is the functionality that was planned to regulate the *Object Found* predicate. As such, we were not able to have that predicate based on sensorial perception, and instead had to apply a similar strategy to what was done in Subsection 5.2.5. A lack of a screen in the MBOT also made it impossible for us to use the GUI functionality for signalling the robot to start the task. With all of these setbacks, and due to time constraints, we had to settle for only running the MDP subtask of the designed FSM instead of the complete FSM. Nonetheless, this FSM will be used

[1] `http://monarch-fp7.eu`, as of September 2014

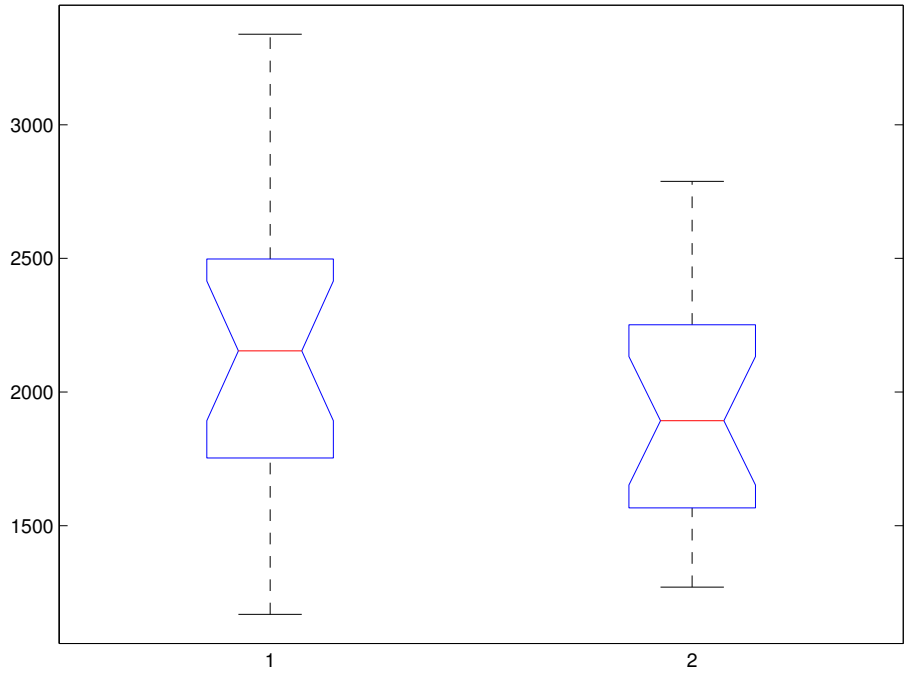


Figure 5.4: One-Way Analysis of Variance Results. The classes are represented in the abscissa axis. Class 1 is resetting the eligibility traces' values. Class 2 is maintaining them. The number of decision episodes is represented in the ordinate axis.

in the RoCKIn Competition 2014.

The domestic environment that we used is the testbed for the SocRob@Home project, and the map that was used can be seen in Figure 5.5 (the segments that are not labeled are not part of the testbed).

A video of the robot accomplishing the task is available in `https://www.youtube.com/watch?v=-KVV2ALqGx8`. Finally, to compare the results that we obtained with a naive approach to the same problem, we contrast their performance with a simple search all over behaviour. This naive approach would comprise visiting every room in a predefined manner, looking for the object in every visit. If the first room to be visited contained the object, this approach would perform as well as the learned MDP approach. In any other situation, one or more rooms would have to additionally be visited in comparison to our approach. As such, we consider our solution to have generally better performance than the naive one.

Figure 5.5: Label Map for the Testbed. The map was first obtained with a real robot, and then the labels were drawn.

Chapter 6

Conclusions

Concluding this thesis, we review its accomplishments and contributions, we provide our conclusions on the application of Reinforcement Learning techniques to domestic robots and, finally, we present possible future work directions.

6.1 Thesis Summary

The main focus of this thesis was to apply Reinforcement Learning techniques to a real robot system. We defined several goals to accomplish this, which we will now review.

The first goal that we defined was to extend the MDM package with RL capabilities. The original goal of MDM was to be an easy-to-use set of tools for creating MDP-based systems. It tackled the issue that these systems are usually hand-tailored for specific problems, resulting in a large amount of implementation-specific work to be done by the system designer. Since RL is based on the MDP framework, it also suffers from those issues. Therefore, we maintained that design philosophy while extending it to support RL problems. In Chapter 3, we explained how we accomplished that extension.

Afterwards, we delineated the task-level software structure for the SocRob@Home robot, which is the application target of this thesis. Despite it being possible to design robotic tasks wholly using DT-based systems, it is unnecessary and cumbersome to do so. Therefore, in Chapter 4, we proposed to use FSMs along with MDPs to define tasks for the robot. Using FSMs allows the definition of the deterministic task segments in a simpler manner than MDPs. Also, given the way that MDM is constructed, it is also possible to implement MDPs as an FSM state. This structure, then, allows for defining tasks that comprehend both stochastic and deterministic segments. Also in this chapter, we describe the simulator that we created for the robot.

Finally, in Chapter 5, we created a Case Study where we used our software structure as well as MDM's RL. Given the SocRob@Home team participation in the RoCKIn competition, we decided to

focus our Case Study in a subtask of one of the three tasks to be performed in it, where we identified uncertainty. The subtask was designed using both FSMs and an MDP, and used RL to cope with the problem's complexity. We also compared our system to a naive one, evidentiating the better performance of our created system.

6.2 Decision-Making in Domestic Robots

In this Subsection, we comment in a non-formal manner on the conclusions we drew while working on this thesis. We start by analyzing the applicability of decision-theoretic methods to domestic robots, moving on to commenting on RL in general.

One of the questions asked in a survey about user needs for domestic robots, (Sung et al., 2009), is whether people generally wanted their robots to be able to make decisions on their own. As the study states, "As much as people spoke of automation of time-consuming tasks, they did not want too much robotic intelligence (also referred to as decision-making power). This was especially true for the tasks that required expert knowledge or involved safety risks. For these tasks, people stressed wanting to work with the robot as opposed to having the robot conduct the entire task.".

It is unlikely that a significant part of the population inquired in the study is familiar with decision-making methods for robotics. For instance, a possible approach to a dialogue system is representing it through a POMDP. However, it does make sense to analyze the applicability of decision-making systems for domestic environments.

During the development of this thesis, we had to find a domestic task suitable to be subject to learning. The frameworks for task learning that we explored concern stochastic environments, which we had difficulties in finding within the domestic setting. If we analyze the FSMs that we present in Appendix B, we find that, despite representing useful tasks for a domestic robot, most of them are fully deterministic (regarding high-level behaviours). Even moving away from the tasks specified for the RoCKIn competition, we had difficulty in finding instances where decision-theoretic methods are useful in domestic tasks.

Regarding task-level learning, it is still possibly dangerous to implement online in a real robot system. The issue of performing a risky action in a certain state can lead to unwanted results and is a safety concern in certain situations. However, offline a priori learning can be useful and risk-free if done in simulation.

Moving on to the comments on RL in general, in Chapter 5, we played the role of an autonomous system designer, which led us to be able to identify the issues behind designing tasks either as discrete system FSMs or as DT based methods.

The fully deterministic FSMs seem to have good software basis for implementation, and give the

designer a sense of control when creating such systems given that its definition is fairly straightforward and simple. However, since the world is not fully deterministic, avoiding uncertainty is mostly unfeasible, making these methods only applicable in certain scenarios. Even if most robotic tasks were not subject to uncertainty, designing large tasks as FSMs would not be practical to an autonomous system designer due to the extensive description that is required to implement such a system.

DT methods, in contrast, despite being able to handle uncertainty, seem to not have good basis for application, and provide a lesser sense of control when compared to designing FSMs, due to their more complex theory. They also suffer from the curse of dimensionality (Bellman, 1957b), which means that the space and time complexity for the optimal solution of these decision-making problems grows exponentially with a linear increase in the cardinality of the problem definition.

It seems, so, that the current state of task-level definition is a strong theoretical basis for designing relatively small tasks, with the additional benefit of also having a basis for having agents learn how to optimally perform said tasks. To increase the size of the tasks that we are able to describe, we have to develop a universal method for knowledge representation. (Minker, 2000) and (Davis and Morgenstern, 2004) cover respectively the past several years of research and the most recent advances in this field.

Having good knowledge representation, however, does not solve the issue of the system designer still having to define every piece of information regarding the system. A possible solution for this, in the DT framework, is to create agents that are able to learn new states. Take, for instance, a robot which mission is to explore another planet. Imagine that the action space of this robot covers every movement possible, which should be relatively feasible. If we were to design the exploration task as a DT based method, and if we had previously mapped a segment of the planet, our state representation would be a discretization of that map. The issue then is: how can the robot explore the rest of the planet without having a previous description of it mapped as states? Some research has been done in the direction of state learning, for instance in (Legenstein et al., 2010), (Jonschkowski and Brock, 2014) and (Jonschkowski and Brock, 2013), where the authors propose methods for learning new states based on previously known states and common knowledge (in this context, we refer to common knowledge as the knowledge on how the world functions, in terms of physics, mathematics, etc.).

An AI that is able to reason over large state spaces, as well as to increase its horizons by, for instance, being able to learn new states, is arguably close to an Artificial General Intelligence - an umbrella term for AI systems that are able to perform any task that a human is. Returning to the standpoint of the system designer, an AGI would most likely not require much designing directly, unless it was created with the purpose of being taught how to act instead of being able to learn by itself. From what we have seen so far, the frameworks that have been created for self-learning are essentially the same as the ones that have been used for learning by imitation (Schaal et al., 1997). Despite this, from the experience taken from the

work developed during this thesis, we believe that the most useful, as well as safe, task-level framework would be to teach agents directly how to act. In (Schaal, 1999), the author explores the application of learning by imitation for teaching robots low-level motor control to perform certain behaviours, reaching the conclusion that it appears to be a promising research direction. We believe that this would also be applicable to high-level decision making, due to the similarity between both.

6.3 Future Work

Given that this thesis covers several different subjects, we examine possible future work on three topics: the SocRob@Home robot (regarding just the task-level domain), the MDM package and Reinforcement Learning in general.

In Chapter 5, we had the issue that not all of the behaviours/actions had been implemented by the time of this thesis. Despite not being exactly task-level, the behaviours for RoCKIn are part of the future work for the robot. Besides that, the obvious future work regarding the robot's task-level domain is to design, for now, all of the tasks required for RoCKIn.

Regarding MDM, we identify several topics which can be improved in the future. The softmax action selection method, for instance, is in principle a better solution than the ε-greedy policy method, given that, instead of selecting random actions, it selects them based on a distribution (it does, however, require setting a parameter called the *temperature*, which in most situations is not obvious how to). Similarly to softmax, the addition of stochastic policies could be a factor in extending MDM's reachability. Hierarchical RL, as well as multi-agent RL, are also possible focus of future work for MDM. Most importantly, perhaps, is the addition of more RL methods, such as, for instance, Dyna (Sutton, 1990) and the more recent R-max (Brafman and Tennenholtz, 2003).

Finally, regarding RL in general, following what we stated in the previous Subsection, we believe that the future work on task-level learning techniques should focus on common sense comprehension and knowledge representation, as well as on the capacity of self-broadening of an agent's horizons. We also believe that, despite the fact that theoretical methods for self-learning and learning by being taught are fairly similar, the research focus should be on the latter, given that it would be more useful for practical application, as well as for society in general.

Appendix A

FSM for the Robótica 2014 Task

Figure A.1 shows the main FSM for the task. Figure A.2 shows the concurrent states *ASK_FOR_ESCORT_OUTSIDE* and *GET_THE_DOOR*.

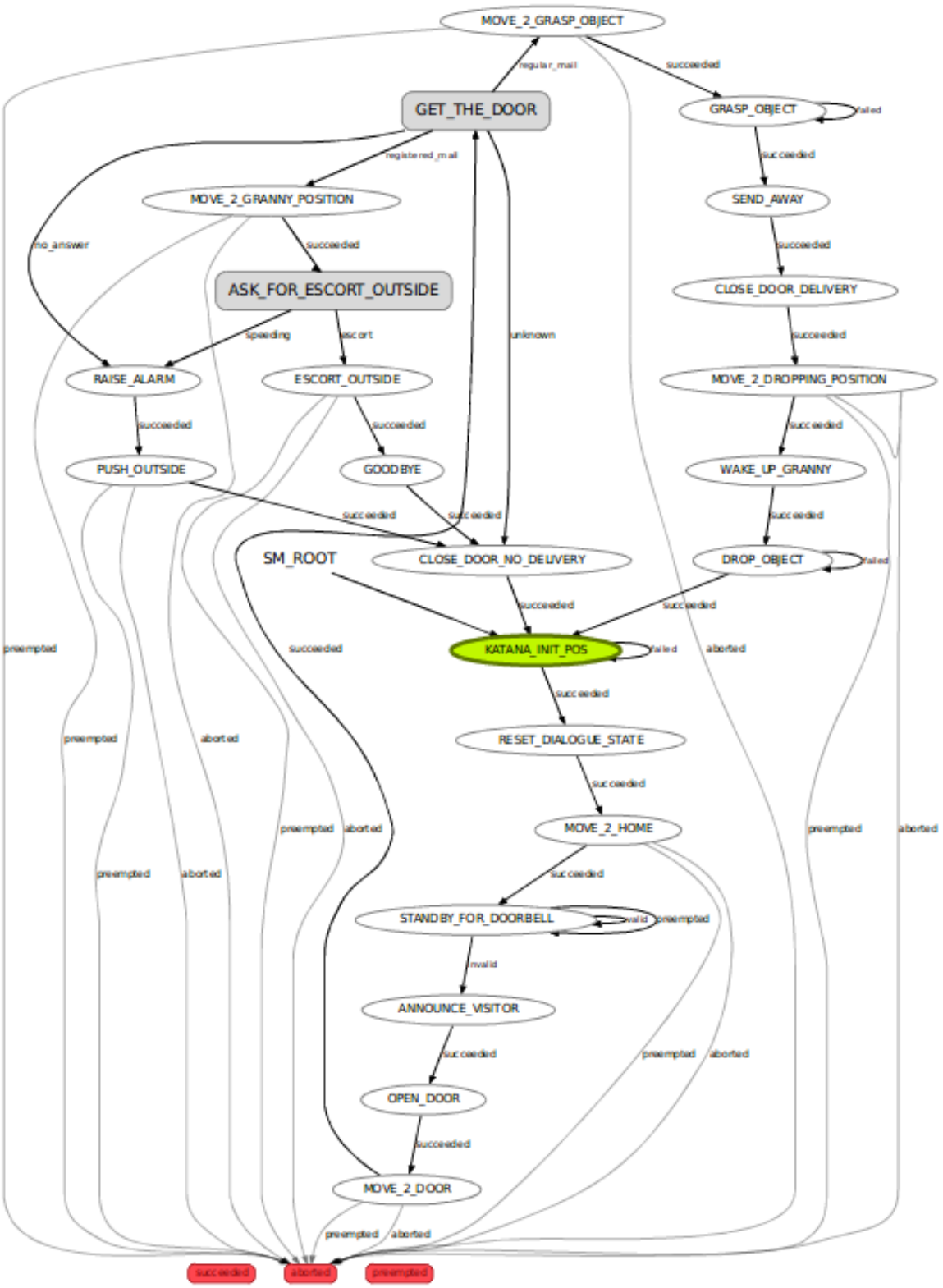
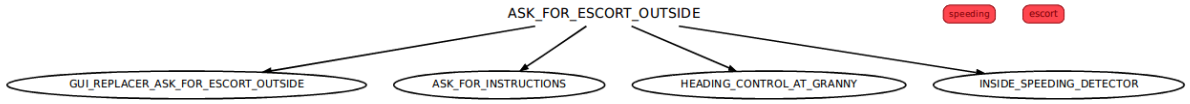
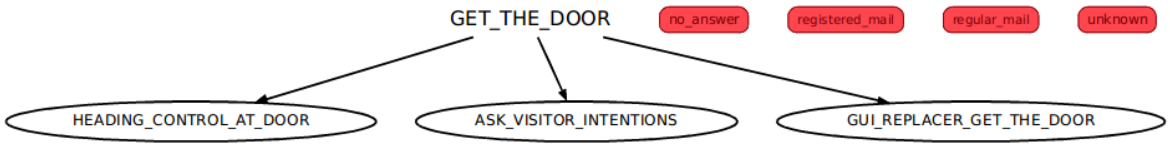


Figure A.1: The FSM for Robótica 2014. The oval shapes represent states, while the rectangular shapes represent concurrent states (see Figure A.2 for their specification). The green state is the initial state and the red states are the termination states.



(a) Concurrent State: *ASK_FOR_ESCORT_OUTSIDE*.



(b) Concurrent State: *GET_THE_DOOR*.

Figure A.2: Concurrent States of the FSM for Robórica 2014.

Appendix B

Pseudo-FSMs of the RoCKIn Tasks

This appendix contains the pseudo-FSMs developed for the RoCKIn competition, which were created by the SocRob@Home team based on the RoCKIn@Home rulebook, (Schneider et al., 2014). A subset of the first task is used for the case-study of the thesis.

B.1 First Task - "Catering for Granny Annie's Comfort"

The first task, which FSM is available in Figure B.1, comprises two sub-tasks: first, perform some action within the house as per the home-owner's request, and then search the house for a home-owner's possession, either with or without prior knowledge.

This thesis' Case Study focuses on the second part of this task, which is a scenario with uncertainty where learning can be useful.

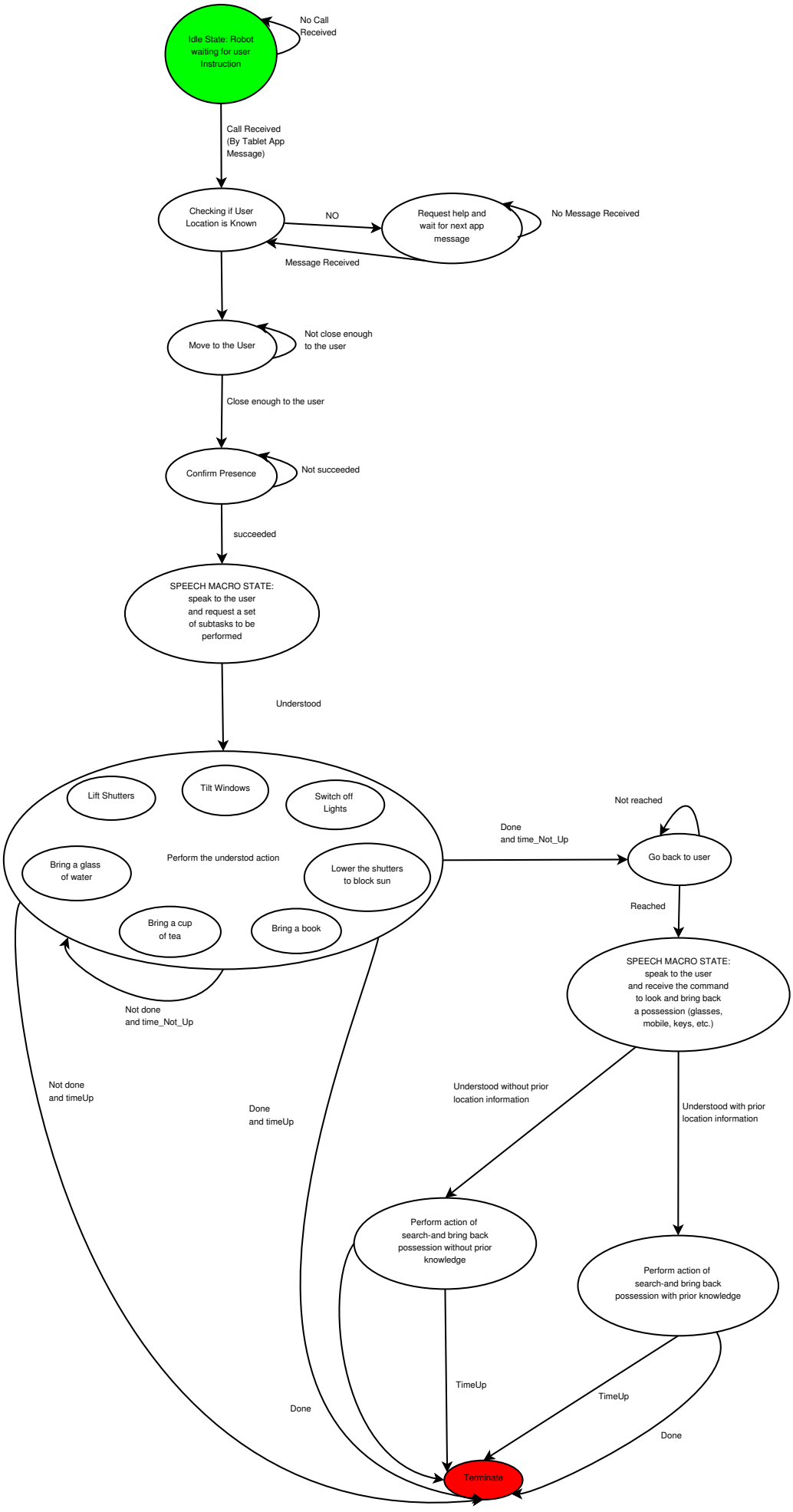


Figure B.1: First RoCKIn Task.

B.2 Second Task - “Welcoming Visitors”

The second task focuses on opening the door for a visitor, identifying him using the intercommunication camera and/or microphone, and behaving according to who he is. There are three possible known visitors and an unknown visitor.

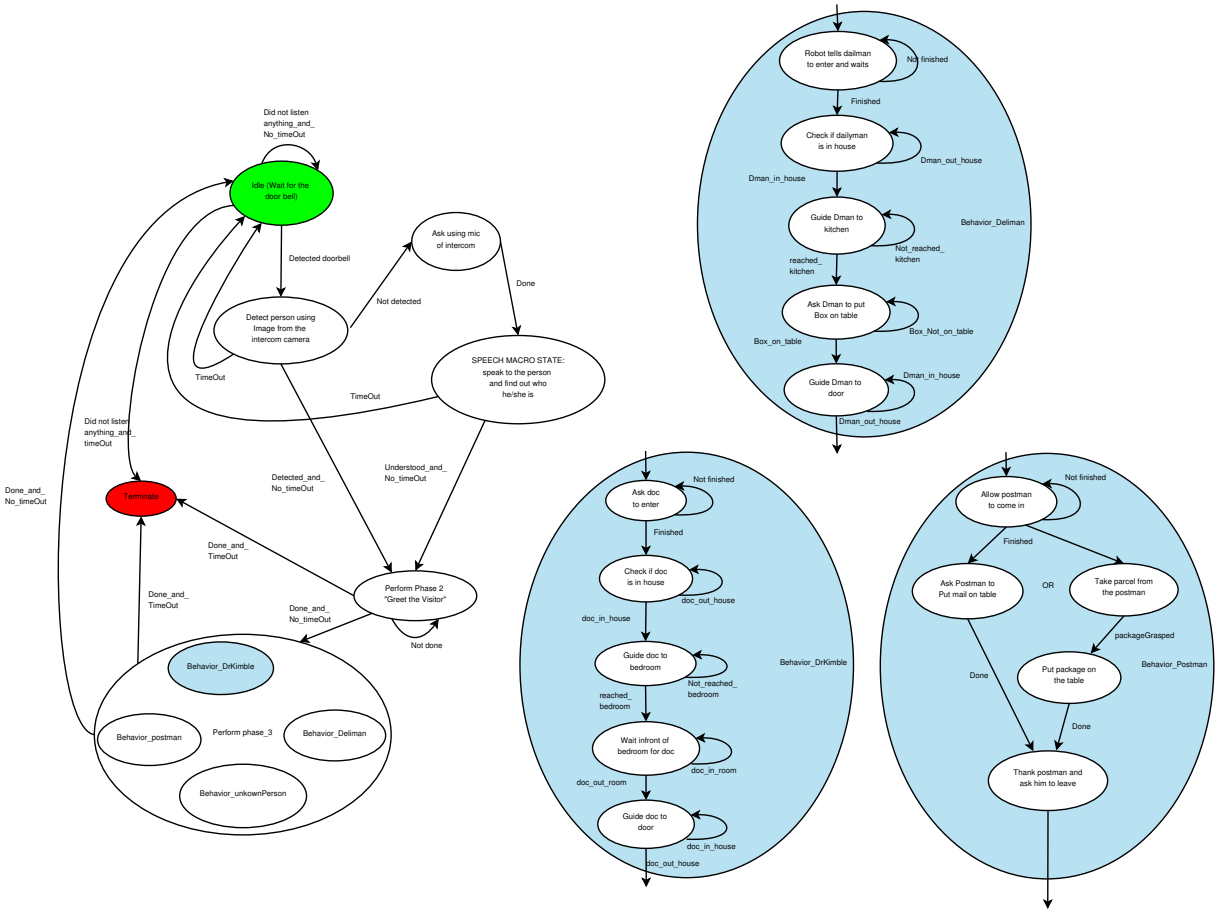


Figure B.2: Second RoCKIn Task.

B.3 Third Task - “Getting to Know my Home”

The third task is meant to affect the first task by changing the previously mentioned prior knowledge of the home-owner’s possession’s location. By interacting with the home-owner, the robot should learn that a certain object’s location has changed and that should affect its reasoning when searching for it.

Although this task is not explored in the Case Study directly, some thought was given on the prior knowledge’s role in the first task.

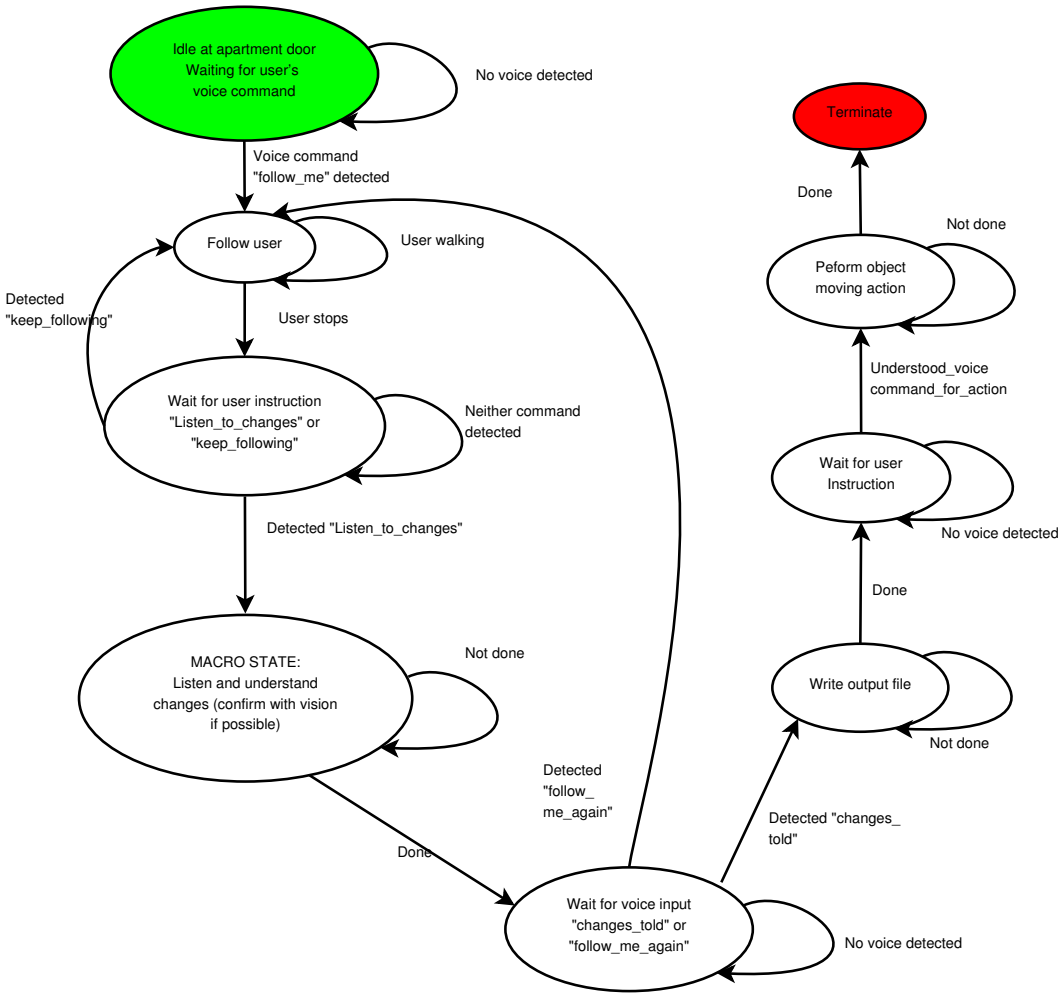


Figure B.3: Third RoCKIn Task.

Appendix C

MDM Package - Tutorial and Example

The purpose of this appendix is to provide an example of using MDM's RL. Given that MDM was first developed in (Messias, 2014), we follow the methodology used in its Appendix B, where the author also provides a usage example. Also, given that the content created here will be used to extend that tutorial, we will only go over the implementation of a learning layer, since the original already describes implementing state, observation, action and control layers. Furthermore, the already existing example is fully implemented and therefore useable. This addition is also fully useable out-of-the-box and will be attached to the already existing example.

C.1 Implementing a Learning Layer

A Learning Layer should be implemented to perform Reinforcement Learning. It substitutes the Control Layer, having the same communication with the Action and State layers as it. The following example shows how to implement a Learning Layer in MDM:

```
#include <ros/ros.h>
#include <mdm_library/sarsa_learning_mdp.h>

using namespace std;
using namespace ros;
using namespace mdm_library;

int main ( int argc, char** argv )
{
    init ( argc, argv, "learning_layer" );

    if ( argc < 5 )
    {
```

```
        ROS_ERROR ( "Usage: rosrun mdm_example demo_control_layer <path to policy
    file> <path to reward file> <path to q values file> <path to eligibility traces
    file>" );
        abort();
    }

    string policy_path = argv[1];
    string reward_path = argv[2];
    string q_values_path = argv[3];
    string eligibility_traces_path = argv[4];

    ALPHA_TYPE alpha = ALPHA_ONE_OVER_T;
    EPSILON_TYPE epsilon = EPSILON_EXP;
    CONTROLLER_TYPE controller = TIMED;

    uint32_t num_states = 6;
    uint32_t num_actions = 4;

    SarsaLearningMDP sarsa ( alpha, epsilon, controller, num_states, num_actions,
    policy_path, reward_path, q_values_path, eligibility_traces_path );

    spin();

    return 0;
}
```

This code instantiates a Learning Layer using SARSA. There are several input arguments to be configured: the policy file path, the reward file path, the Q-Values file path and the eligibility traces file path. All of these correspond to files with Boost's[1] matrix/vector default configuration. For instance, a policy vector file could be:

```
[6](0,0,0,1,2,0)
```

and a reward matrix file could be:

```
[6,4]((-1,-1,-1,-1),(-1,-1,-1,-1),(100,-1,-1,-1),(-1,-1,-1,-1),(-1,-1,-1,-1)
    ,(-1,-1,-1,-1))
```

Providing an eligibility traces file is not required, and there is another constructor that does not require it. We give the option to initialize the eligibility traces from a file so that a previous run can be recuperated.

[1] `http://www.boost.org`, as of September 2014

Besides the input arguments, instantiating the SARSA class also requires five more arguments: the α type, the ε type, the controller type, the number of states and the number of actions.

The first two define whether those algorithm parameters are constant or functions of time. The α type can be: *ALPHA_CONSTANT*, *ALPHA_ONE_OVER_T* ($\alpha(t) = 1/t$) and *ALPHA_ONE_OVER_T_SQUARED* ($\alpha(t) = 1/t^2$). The ε type can be: *EPSILON_CONSTANT*, *EPSILON_ONE_OVER_T* ($\varepsilon(t) = 1/t$), *EPSILON_ONE_OVER_T_SQUARED* ($\varepsilon(t) = 1/t^2$), *EPSILON_ONE_OVER_T_ROOTED* ($\varepsilon(t) = 1/\sqrt{t}$) and *EPSILON_EXP* ($\varepsilon(t) = e^{-50t}$). In the context of MDM, time t is discretized in decision episodes. The file *mdm_library/include/mdm_library/learning_defs.h* can be modified to include more functions or change the already existing ones.

The last three arguments are specific to MDM. The controller type defines whether the controller used when learning operates on a timed basis or on an event basis (where an event is the perception of a new state). Therefore, the possible controller types are *EVENT* and *TIMED*. The number of actions and the number of states correspond to those of the problem definition.

Besides the arguments, several ROS parameters also have to be defined. If the arguments α type and ε type are set to constant, the parameters "alpha" and "epsilon" are interpreted by MDM as their constant value. The other Reinforcement Learning parameters, "gamma" and "lambda", are also defined as ROS parameters. Besides those, there are only three more parameters: *policy_update_frequency*, *impossible_action_reward* and *reward_type*. The former tells MDM the interval with which to update and save the policy to file, and to save the Q-Values table and the eligibility traces table (if being used) to file. The second refers to the reward value that should be given to the agent when it tries to perform an action which is impossible to realize in the current state. The latter tells MDM whether the provided reward model is vector or matrix based.

If instead of SARSA, the goal is use Q-Learning, the following usage is recommended:

```
#include <ros/ros.h>
#include <mdm_library/q_learning_mdp.h>

using namespace std;
using namespace ros;
using namespace mdm_library;

int main ( int argc, char** argv )
{
    init ( argc, argv, "learning_layer" );

    if ( argc < 5 )
    {
```

```
        ROS_ERROR ( "Usage: rosrun mdm_example demo_q_learning <path to behaviour
    policy file> <path to learning policy file> <path to reward file> <path to q
    values file> <path to eligibility traces file>" );
        abort();
    }

    string policy_path = argv[1];
    string reward_path = argv[2];
    string q_values_path = argv[3];
    string eligibility_traces_path = argv[4];

    ALPHA_TYPE alpha = ALPHA_ONE_OVER_T;
    EPSILON_TYPE epsilon = EPSILON_ONE_OVER_T;
    CONTROLLER_TYPE controller = EVENT;

    uint32_t num_states = 6;
    uint32_t num_actions = 4;

    QLearningMDP ql (alpha, epsilon, controller, num_states, num_actions,
    policy_path, reward_path, q_values_path, eligibility_traces_path );

    spin();

    return 0;
}
```

The expected output and result of using MDM's Reinforcement Learning is a learned policy. MDM also supports saving the Q-Values file for providing a starting point if something goes wrong while the algorithms are running. This file is saved every *policy_update_frequency* decision episodes and is automatically loaded when a Learning Layer is instantiated. If it is empty, MDM will initialize it as zeros. If, however, it is not empty, MDM will load the current Q-Values and use them as starting conditions for learning.

Appendix D

Case Study Results

In this appendix we present the results that were obtained during the Case Study, namely the SARSA(λ) and Q(λ) results that were integrated with the FSM in the final result. It can be observed that there are several states that seemingly remained unexplored. The reason for this is that those states never actually occur; for instance, given that we always consider the person to be in the dining room, there is never the situation where the robot is in any room that is not the dining room and that the person is found.

D.1 SARSA(λ)

$$Q(s,a) = \begin{array}{c} \\ s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \\ s_8 \\ s_9 \\ s_{10} \\ s_{11} \\ s_{12} \\ s_{13} \\ s_{14} \\ s_{15} \\ s_{16} \\ s_{17} \\ s_{18} \\ s_{19} \\ s_{20} \\ s_{21} \\ s_{22} \\ s_{23} \end{array} \begin{array}{c} \begin{array}{cccccc} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \end{array} \\ \begin{pmatrix} -9.4756 & -4.6864 & -9.77818 & -7.68756 & -10.0539 & -7.40477 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -7.2791 & -2.31796 & 11.1487 & 21.3748 & 23.7 & -7.09132 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -9.52684 & -8.93964 & -10.1853 & -10.7153 & 21.1652 & -9.35748 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -6.9558 & 22.072 & -8.36097 & -7.67316 & 11.8196 & 15.621 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 18.7115 & 34.5057 & -6.06314 & -4.45234 & 27.1839 & 4.59909 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 10.898 & 28.5216 & 10.8481 & -8.00167 & 20.8339 & 10.2454 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -9.64135 & 1.75437 & -9.62134 & -9.84935 & 6.96057 & -9.58758 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -8.15017 & -6.87386 & -10.2852 & -7.89017 & -7.24236 & -9.53799 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$
Q(s,a) = \begin{array}{c} \\ s_{24} \\ s_{25} \\ s_{26} \\ s_{27} \\ s_{28} \\ s_{29} \\ s_{30} \\ s_{31} \\ s_{32} \\ s_{33} \\ s_{34} \\ s_{35} \\ s_{36} \\ s_{37} \\ s_{38} \\ s_{39} \\ s_{40} \\ s_{41} \\ s_{42} \\ s_{43} \\ s_{44} \\ s_{45} \\ s_{46} \\ s_{47} \end{array}
\begin{array}{c}
\begin{array}{cccccc} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \end{array} \\
\left(\begin{array}{cccccc}
-9.49732 & 18.6523 & -9.2732 & -6.61753 & -9.31567 & 9.03183 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
-7.83543 & 16.0607 & -8.02131 & -7.80803 & -7.56115 & -8.02835 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
9.08102 & -8.90568 & 10.0321 & 5.02036 & -8.74584 & -9.01039 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -0.617653 & 0 & 0 & 0 \\
19.8559 & 21.9477 & 5.21974 & 38.8954 & 52.483 & 57.9103 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}\right)
\end{array}
$$

$$Q(s,a) = \begin{array}{c} \\ s_{48} \\ s_{49} \\ s_{50} \\ s_{51} \\ s_{52} \\ s_{53} \\ s_{54} \\ s_{55} \\ s_{56} \\ s_{57} \\ s_{58} \\ s_{59} \\ s_{60} \\ s_{61} \\ s_{62} \\ s_{63} \\ s_{64} \\ s_{65} \\ s_{66} \\ s_{67} \\ s_{68} \\ s_{69} \\ s_{70} \\ s_{71} \end{array} \begin{array}{c} \begin{array}{cccccc} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \end{array} \\ \begin{pmatrix} 2.71079 & -2.56404 & 15.6833 & 11.0922 & 7.17155 & 0.744427 \\ 0 & 0 & 7.2523 & 0 & 0 & 0 \\ 1.98983 & 3.57166 & 8.81448 & -0.560235 & 36.2678 & -8.52143 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 10.5276 & 41.2531 & 12.4217 & 4.89155 & 42.2284 & -6.92786 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -8.13684 & 43.7363 & 14.0641 & 34.7932 & 26.352 & 4.15018 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -7.4734 & 47.832 & 8.23436 & -4.7689 & 31.989 & -3.56277 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -8.21299 & 44.6805 & -7.13058 & -0.966889 & 26.6807 & 20.1038 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 3.21673 & -9.11738 & 10.6532 & -3.58457 & -0.0888469 & -8.75732 \\ -6.86086 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2.24852 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 7.21236 & 40.9424 & 15.3504 & 19.3322 & 26.9197 & 18.8612 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6.64394 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$\pi(s) = \begin{array}{c} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \\ s_8 \\ s_9 \\ s_{10} \\ s_{11} \\ s_{12} \\ s_{13} \\ s_{14} \\ s_{15} \\ s_{16} \\ s_{17} \\ s_{18} \\ s_{19} \\ s_{20} \\ s_{21} \\ s_{22} \\ s_{23} \end{array} \begin{pmatrix} 1 \\ 0 \\ 4 \\ 0 \\ 4 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 4 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{array}{c} s_{24} \\ s_{25} \\ s_{26} \\ s_{27} \\ s_{28} \\ s_{29} \\ s_{30} \\ s_{31} \\ s_{32} \\ s_{33} \\ s_{34} \\ s_{35} \\ s_{36} \\ s_{37} \\ s_{38} \\ s_{39} \\ s_{40} \\ s_{41} \\ s_{42} \\ s_{43} \\ s_{44} \\ s_{45} \\ s_{46} \\ s_{47} \end{array} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 5 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{array}{c} s_{48} \\ s_{49} \\ s_{50} \\ s_{51} \\ s_{52} \\ s_{53} \\ s_{54} \\ s_{55} \\ s_{56} \\ s_{57} \\ s_{58} \\ s_{59} \\ s_{60} \\ s_{61} \\ s_{62} \\ s_{63} \\ s_{64} \\ s_{65} \\ s_{66} \\ s_{67} \\ s_{68} \\ s_{69} \\ s_{70} \\ s_{71} \end{array} \begin{pmatrix} 2 \\ 2 \\ 4 \\ 0 \\ 4 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 2 \\ 1 \\ 0 \\ 0 \\ 4 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

D.2 Q(λ)

$$Q(s,a) = \begin{array}{c} \\ s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \\ s_8 \\ s_9 \\ s_{10} \\ s_{11} \\ s_{12} \\ s_{13} \\ s_{14} \\ s_{15} \\ s_{16} \\ s_{17} \\ s_{18} \\ s_{19} \\ s_{20} \\ s_{21} \\ s_{22} \\ s_{23} \end{array} \begin{array}{c} \begin{array}{cccccc} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \end{array} \\ \begin{pmatrix} 0 & -0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.75 & 0 & 0 & 13.3211 & 52.7086 & 32.3051 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 13.4851 & 0 & 13.3601 & 26.9255 & 46.2234 & 17.7837 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 33.9619 & 71.2718 & 25.1342 & 33.9619 & 25.5838 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 17.9645 & 58.3664 & 39.3305 & 37.091 & 24.4981 & 32.8634 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.0597114 & 0 & -0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 12.5214 & 0 & 0 & 4.75965 & -0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$
Q(s,a) = \begin{array}{c} \\ s_{24} \\ s_{25} \\ s_{26} \\ s_{27} \\ s_{28} \\ s_{29} \\ s_{30} \\ s_{31} \\ s_{32} \\ s_{33} \\ s_{34} \\ s_{35} \\ s_{36} \\ s_{37} \\ s_{38} \\ s_{39} \\ s_{40} \\ s_{41} \\ s_{42} \\ s_{43} \\ s_{44} \\ s_{45} \\ s_{46} \\ s_{47} \end{array}
\begin{array}{c}
\begin{array}{cccccc} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \end{array} \\
\left(\begin{array}{cccccc}
-0.5 & 11.2389 & 0 & -0.5 & 0 & -0.309711 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
3.00643 & 39.1548 & 7.87887 & 12.2871 & 15.5895 & 16.6952 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
-0.5 & -0.5 & 7.25599 & 2.22401 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
38.2943 & 75.3579 & 56.6557 & 44.1488 & 53.3644 & 113.464 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}\right)
\end{array}
$$

$$Q(s,a) = \begin{array}{c} \\ s_{48} \\ s_{49} \\ s_{50} \\ s_{51} \\ s_{52} \\ s_{53} \\ s_{54} \\ s_{55} \\ s_{56} \\ s_{57} \\ s_{58} \\ s_{59} \\ s_{60} \\ s_{61} \\ s_{62} \\ s_{63} \\ s_{64} \\ s_{65} \\ s_{66} \\ s_{67} \\ s_{68} \\ s_{69} \\ s_{70} \\ s_{71} \end{array} \begin{array}{c} \begin{array}{cccccc} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \end{array} \\ \begin{pmatrix} -0.104976 & 9.30421 & 39.5805 & 14.9625 & 3.97932 & 6.32643 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -0.5 & -0.5 & 14.6765 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.5 & 0 & 0 & -0.5 & 13.8669 & -0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 7.97262 & 96.852 & 43.1511 & 23.5437 & 43.883 & 62.0956 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 57.9478 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 12.5589 & 43.8476 & 0 & 21.2422 & 21.8192 & 21.8192 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 11.073 & -0.5 & -0.5 & -0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 33.1488 & 45.9771 & 0 & 11.8849 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$\pi(s) = \begin{array}{c} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \\ s_8 \\ s_9 \\ s_{10} \\ s_{11} \\ s_{12} \\ s_{13} \\ s_{14} \\ s_{15} \\ s_{16} \\ s_{17} \\ s_{18} \\ s_{19} \\ s_{20} \\ s_{21} \\ s_{22} \\ s_{23} \end{array} \begin{pmatrix} 0 \\ 0 \\ 4 \\ 0 \\ 4 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{array}{c} s_{24} \\ s_{25} \\ s_{26} \\ s_{27} \\ s_{28} \\ s_{29} \\ s_{30} \\ s_{31} \\ s_{32} \\ s_{33} \\ s_{34} \\ s_{35} \\ s_{36} \\ s_{37} \\ s_{38} \\ s_{39} \\ s_{40} \\ s_{41} \\ s_{42} \\ s_{43} \\ s_{44} \\ s_{45} \\ s_{46} \\ s_{47} \end{array} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 5 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{array}{c} s_{48} \\ s_{49} \\ s_{50} \\ s_{51} \\ s_{52} \\ s_{53} \\ s_{54} \\ s_{55} \\ s_{56} \\ s_{57} \\ s_{58} \\ s_{59} \\ s_{60} \\ s_{61} \\ s_{62} \\ s_{63} \\ s_{64} \\ s_{65} \\ s_{66} \\ s_{67} \\ s_{68} \\ s_{69} \\ s_{70} \\ s_{71} \end{array} \begin{pmatrix} 2 \\ 0 \\ 4 \\ 0 \\ 4 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

References

Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y Ng. An application of reinforcement learning to aerobatic helicopter flight. *Advances in neural information processing systems*, 19:1, 2007.

Andrew G Barto and Michael Duff. Monte Carlo matrix inversion and reinforcement learning. *Advances in Neural Information Processing Systems*, pages 687–687, 1994.

Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.

Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Knightcap: A chess program that learns by combining TD(λ) with game-tree search. In *Proceedings of the 15th International Conference on Machine Learning*, 1998.

R. Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6, 1957a.

R. Bellman. Dynamic Programming, 1957b.

Daniel S Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, 27(4):819–840, 2002.

Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1996.

George E. P. Box et al. Some Theorems on Quadratic Forms Applied in the Study of Analysis of Variance Problems, I. Effect of Inequality of Variance in the One-Way Classification. *The Annals of Mathematical Statistics*, 25(2):290–302, 1954.

Ronen I. Brafman and Moshe Tennenholtz. R-max - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning. *The Journal of Machine Learning Research*, 3:213–231, 2003.

Jennifer Casper and Robin R. Murphy. Human-robot interactions during the robot-assisted urban search and rescue response at the World Trade Center. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 33(3):367–385, 2003.

Christos G Cassandras and Stephane Lafortune. *Introduction to discrete event systems*. Springer, 2008.

Caroline Claus and Craig Boutilier. The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems. In *AAAI/IAAI*, pages 746–752, 1998.

Kerstin Dautenhahn, Sarah Woods, Christina Kaouri, Michael L Walters, Kheng Lee Koay, and Iain Werry. What is a robot companion-friend, assistant or butler? In *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 1192–1197. IEEE, 2005.

B Davies. A review of robotics in surgery. *Proceedings of the Institution of Mechanical Engineers, Part H: Journal of Engineering in Medicine*, 214(1):129–140, 2000.

Ernest Davis and Leora Morgenstern. Introduction: Progress in formal commonsense reasoning. *Artificial Intelligence*, 153(1):1–12, 2004.

Thomas G Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

Peggy Fidelman and Peter Stone. Learning Ball Acquisition on a Physical Robot. In *2004 International Symposium on Robotics and Automation (ISRA)*, page 6, 2004.

Bill Gates. A Robot in Every Home. *Scientific American*, 296(1):58–65, 2007.

Masato Hirose and Kenichi Ogawa. Honda humanoid robots development. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 365(1850):11–19, 2007.

Javier Ibañez-Guzmán, Xu Jian, Andrew Malcolm, Zhiming Gong, Chun Wah Chan, and A Tay. Autonomous armoured logistics carrier for natural environments. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 1, pages 473–478. IEEE, 2004.

Rico Jonschkowski and Oliver Brock. Learning task-specific state representations by maximizing slowness and predictability. In *Proceedings of the 6th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems (ERLARS)*, 2013.

Rico Jonschkowski and Oliver Brock. State Representation Learning in Robotics: Using Prior Knowledge about Physical Interaction. In *Proceedings of Robotics: Science and Systems*, 2014.

Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

Robert Legenstein, Niko Wilbert, and Laurenz Wiskott. Reinforcement Learning on Slow Features of High-Dimensional Input Streams. *PLoS computational biology*, 6(8):e1000894, 2010.

Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *ICML*, volume 94, pages 157–163, 1994.

Manja Lohse, Frank Hegel, and Britta Wrede. Domestic Applications for Social Robots-an online survey on the influence of appearance and capabilities. *Journal of Physical Agents*, 2(2):21–32, 2008.

Edwin Mandfield. The diffusion of industrial robots in Japan and the United States. *Research Policy*, 18 (4):183–192, 1989.

Donald W Marquardt. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *Journal of the Society for Industrial & Applied Mathematics*, 11(2):431–441, 1963.

Christian Martens, Oliver Prenzel, and Axel Gräser. *The Rehabilitation Robots FRIEND-I & II:Ddaily Life Independency Through Semi-Autonomous Task-Execution*. Citeseer, 2007.

S Maurice, RC Wiens, M Saccoccio, B Barraclough, O Gasnault, O Forni, N Mangold, D Baratoux, S Bender, G Berger, et al. The ChemCam instrument suite on the Mars Science Laboratory (MSL) rover: science objectives and mast unit description. *Space science reviews*, 170(1-4):95–166, 2012.

João Vicente Teixeira de Sousa Messias. *Decision-Making under Uncertainty for Real Robot Teams*. PhD thesis, Instituto Superior Técnico, 2014.

Jack Minker. *Logic-based Artificial Intelligence*. Springer, 2000.

Marvin Minsky. Why People Think Computers Can't. *AI Magazine*, 3, 1982.

G. E. Monahan. A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science*, 28:1 – 16, 1982.

John Moody and Matthew Saffell. Learning to trade via direct reinforcement. *Neural Networks, IEEE Transactions on*, 12(4):875–889, 2001.

Bojan Nemec and Ales Ude. Reinforcement learning of ball-in-a-cup playing robot. In *Robotics and Biomimetics (ROBIO), 2011 IEEE International Conference on*, pages 2682–2987. IEEE, 2011.

Jordi Palacin, José Antonio Salse, Ignasi Valgañón, and Xavi Clua. Building a mobile robot for a floor-cleaning operation in domestic environments. *IEEE Transactions on Instrumentation and Measurement*, 53(5):1418–1424, 2004.

Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, pages 1043–1049, 1998.

Jing Peng and Ronald J Williams. Incremental multi-step Q-learning. *Machine Learning*, 22(1-3):283–290, 1996.

Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, Inc., 1994.

Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, volume 3, page 5, 2009.

Gavin A Rummery and Mahesan Niranjan. On-line Q-learning using connectionist systems. Technical report, University of Cambridge, Department of Engineering, 1994.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A modern approach*. Pearson, third edition, 2003.

Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6): 233–242, 1999.

Stefan Schaal et al. Learning from demonstration. *Advances in neural information processing systems*, pages 1040–1046, 1997.

Stefan Schiffer, Alexander Ferrein, and Gerhard Lakemeyer. Caesar: an intelligent domestic service robot. *Intelligent Service Robotics*, 5(4):259–273, 2012.

Sven Schneider, Gerhard Kraetzschmar, Aamir Ahmad, Francesco Amigoni, Iman Awaad, Jakob Berghofer, Rainer Bischoff, Andrea Bonarini, Rhama Dwiputra, Giulio Fontana, Pedro Lima, et al. RoCKIn@ Home, 2014.

Satinder P Singh and Richard S Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1-3):123–158, 1996.

Satinder P Singh, Tommi Jaakkola, and Michael I Jordan. Learning Without State-Estimation in Partially Observable Markovian Decision Processes. In *ICML*, pages 284–292, 1994.

Peter Stone, Richard S Sutton, and Gregory Kuhlmann. Reinforcement Learning for RoboCup Soccer Keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

JaYoung Sung, Henrik Iskov Christensen, and Rebecca E Grinter. Sketching the future: Assessing user needs for domestic robots. In *Robot and Human Interactive Communication, 2009. RO-MAN 2009. The 18th IEEE International Symposium on*, pages 153–158, 2009.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1): 9–44, 1988.

Richard S Sutton. Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In *ML*, pages 216–224, 1990.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181 – 211, 1999.

Ming Tan. Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents. In *Proceedings of the tenth international conference on machine learning*, volume 337, 1993.

Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3): 58–68, 1995.

Sebastian B Thrun. Efficient exploration in reinforcement learning. Technical report, Carnegie-Mellon University, 1992.

Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards.* PhD thesis, University of Cambridge, 1989.