MARCELO LI KOGA

Relational Transfer across Reinforcement Learning Tasks
via Abstract Policies

São Paulo
2014

MARCELO LI KOGA

Relational Transfer across Reinforcement Learning Tasks via Abstract Policies

Dissertation submitted to Escola Politécnica
da Universidade de São Paulo in partial
fulfillment of the requirements for the degree
of Master of Science

Area of research:
Computer Engineering

São Paulo
2014

MARCELO LI KOGA

Relational Transfer across Reinforcement Learning Tasks via Abstract Policies

Dissertation submitted to Escola Politécnica
da Universidade de São Paulo in partial
fulfillment of the requirements for the degree
of Master of Science

Area of research:
Computer Engineering

Advisor: Full Professor Anna Helena Reali
Costa

São Paulo
2014

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 14 de janeiro de 2014.

Assinatura do autor

Assinatura do orientador

FICHA CATALOGRÁFICA

Koga, Marcelo Li

Relational transfer across reinforcement learning tasks via abstract policies / M.L. Koga. – versão corr. -- São Paulo, 2014. 119 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Inteligência artificial 2.Aprendizado computacional 4.Processos de Markov 4.Representação de conhecimento I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Anna Helena Reali Costa. She has given me essential guidance for shaping the research and encouraged me to achieve to the best of my ability.

I also would like to express my very great appreciation to prof. Valdinei Freire da Silva, who contributed a lot to making my research possible.

I would like to offer my special thanks to my family, especially my mother, Helena; my father, Sérgio; and my sister, Letícia, for their support and encouragement.

I wish to acknowledge the help provided by prof. Fábio Cozman, with advice and assistance in writing papers.

I also thank all the members of LTI (Laboratório de Técnicas Inteligentes - USP) for valuable discussions and comments regarding my research.

My special thanks are extended to my co-workers and friends, Luiz Lamardo and Laio Burim, and also to all my friends, for their support and understanding.

I also gratefully acknowledge financial support from FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo).

Finally, I thank God.

"The learning and knowledge that we have, is, at the most, but little compared to that of which we are ignorant"

Plato

RESUMO

Na construção de agentes inteligentes para a solução de problemas de decisão sequenciais, o uso de aprendizado por reforço é necessário quando o agente não possui conhecimento suficiente para construir um modelo completo do problema. Entretanto, o aprendizado de uma política ótima é em geral muito lento pois deve ser atingido através de tentativa-e-erro e de repetidas interações do agente com o ambiente. Uma das técnicas para se acelerar esse processo é possibilitar a transferência de aprendizado, ou seja, utilizar o conhecimento adquirido para se resolver tarefas passadas no aprendizado de novas tarefas. Assim, se as tarefas tiverem similaridades, o conhecimento prévio guiará o agente para um aprendizado mais rápido. Neste trabalho é explorado o uso de uma representação relacional, que explicita relações entre objetos e suas propriedades. Essa representação possibilita que se explore abstração e semelhanças estruturais entre as tarefas, possibilitando a generalização de políticas de ação para o uso em tarefas diferentes, porém relacionadas. Este trabalho contribui com dois algoritmos livres de modelo para construção online de políticas abstratas: AbsSarsa(λ) e AbsProb-RL. O primeiro constrói uma política abstrata determinística através de funções-valor, enquanto o segundo constrói uma política abstrata estocástica através de busca direta no espaço de políticas. Também é proposta a arquitetura S2L-RL para o agente, que possui dois níveis de aprendizado: o nível abstrato e o nível concreto. Uma política concreta é construída simultaneamente a uma política abstrata, que pode ser utilizada tanto para guiar o agente no problema atual quanto para guiá-lo em um novo problema futuro. Experimentos com tarefas de navegação robótica mostram que essas técnicas são efetivas na melhoria do desempenho do agente, principalmente nas fases iniciais do aprendizado, quando o agente desconhece completamente o novo problema.

Palavras-chave: inteligência artificial, processos de Markov, aprendizado computacional relacional, representação do conhecimento.

ABSTRACT

When designing intelligent agents that must solve sequential decision problems, often we do not have enough knowledge to build a complete model for the problems at hand. Reinforcement learning enables an agent to learn behavior by acquiring experience through trial-and-error interactions with the environment. However, knowledge is usually built from scratch and learning the optimal policy may take a long time. In this work, we improve the learning performance by exploring transfer learning; that is, the knowledge acquired in previous source tasks is used to accelerate learning in new target tasks. If the tasks present similarities, then the transferred knowledge guides the agent towards faster learning. We explore the use of a relational representation that allows description of relationships among objects. This representation simplifies the use of abstraction and the extraction of the similarities among tasks, enabling the generalization of solutions that can be used across different, but related, tasks. This work presents two model-free algorithms for online learning of abstract policies: AbsSarsa(λ) and AbsProb-RL. The former builds a deterministic abstract policy from value functions, while the latter builds a stochastic abstract policy through direct search on the space of policies. We also propose the S2L-RL agent architecture, containing two levels of learning: an abstract level and a ground level. The agent simultaneously builds a ground policy and an abstract policy; not only the abstract policy can accelerate learning on the current task, but also it can guide the agent in a future task. Experiments in a robotic navigation environment show that these techniques are effective in improving the agent's learning performance, especially during the early stages of the learning process, when the agent is completely unaware of the new task.

Keywords: artificial intelligence, Markov processes, relational machine learning, knowledge representation.

LIST OF FIGURES

2.1	<i>The perception-action cycle</i> – Agent observes current state from environment, choose an action to execute, receives reward and observes new state.	30
3.1	Three characteristics an agent with transfer learning might show when compared to an agent without transfer. Source: Torrey and Shavlik (2009)	47
3.2	Simple navigation environment with 5 rooms, numbered 1 to 5.	51
4.1	Comparison of policy values between learning in the ground level with the plain Sarsa(λ) algorithm (value of ground policy shown), and the approaches to active and passive abstraction with AbsSarsa(λ) (value of abstract policies shown).	65
4.2	Simple discrete environment. Each cell represents a state; cell 4 is the goal and arrows indicate optimal policy.	67
4.3	<i>Source tasks</i> Ω_1 and Ω_2 – white cells are dirt; light grey, grass; dotted pattern, sand; and dark grey, rocks. Thick black lines represent walls, the initial state is the top-left cell and the target is marked with circles.	77
4.4	<i>Transferred knowledge</i> – Optimal policies for the source tasks (π_1 for Ω_1 and π_2 for Ω_2); π_g is a generalized policy that suboptimally satisfies both Ω_1 and Ω_2 . Actually, the policies are stochastic, but the arrows just represent the most likely action of each state.	77
4.5	<i>Target tasks</i> Ω_3 and Ω_4 – Tasks to be learned with the aid of past policies (π_1 and π_2 , or π_g). Arrows indicate the optimal action for each state.	78
5.1	The two-layered learning agent architecture	82
5.2	<i>Exploration strategies</i> – Apart from exploring with π_{random} and exploiting $\pi_{current}$, ψ -reuse adds one more policy to be exploited, π_{past} .	83
6.1	<i>Navigation Domain</i> $\mathcal{D}_1$ – Thick lines represent walls; darker cells, rooms; and white cells, corridors. Discontinuities in the walls represent doors. Each numbered cell represents a ground state. Cell number 4 is the goal, as an example; the states considered near the goal are also marked in darker colors.	90

6.2	<i>Domain $\mathcal{D}_2$</i> – Thick lines represent walls and doors are marked with dn , $n \in \{1, \dots, 11\}$. Each numbered cell represents a ground state. There is only one room marker per room. They are represented by rm , $m \in \{1, \dots, 11\}$. Corridor markers are omitted.	95
6.3	Value of abstract policy built with AbsProb-RL. With $\mu_G(i) = 1$, three different estimation window functions $H(i)$ are compared.	96
6.4	Value of abstract policy built with AbsProb-RL. With $H(i) = 20i$, three different learning rates $\mu_G(i)$ are compared.	97
6.5	Value of abstract policy built with AbsProb-RL. With $\mu_G(i) = 1/i$, four different estimation window functions $H(i)$ are compared.	98
6.6	Comparison between Sarsa and S2L-RL with AbsSarsa(λ). S2L-RL simultaneously learns an abstract and a ground policy and can also reuse past abstract policies to accelerate learning.	99
6.7	Evolution of policy usage in S2L-RL, showing the average usage of each policy for 150 executions of S2L-RL.	100
6.8	<i>Navigation Domain $\mathcal{D}_3$</i> – Black areas represent walls; darker cells, rooms; lighter cells, corridors and doors are marked with ‘d’. Each cell represents one state, except for wall and door cells. The goals of each source task are marked with ‘S’ and the goals of each target task with ‘T’.	101
6.9	<i>Transfer results for 20 source tasks</i> – Performance of the transfer of a generalized policy using knowledge of 20 source tasks, compared with transfer of policy libraries ($L0$, $L25$, $L50$, and $L100$) and Q -Learning without past knowledge (no transfer). Each point represents the average value of 100 executions and the error bars indicate the 95% confidence intervals based on the Student’s t -distribution.	103
6.10	<i>Evolution of size of source task set</i> – Performance after 1000 episodes for different numbers of source tasks. Each point represents the average value of 100 executions.	105
6.11	<i>Navigation domain $\mathcal{D}_4$</i> – 10 target tasks to evaluate transfer among tasks of different domains.	106
6.12	<i>Results of transfer to new domain</i> – Performance of policy reuse methods applying knowledge of 20 source tasks in a new domain. Each point represents the average value of 100 executions and the error bars indicate the 95% confidence intervals based on the Student’s t -distribution.	107

LIST OF TABLES

3.1	Transfer learning methods discussed in Section 3.2, classifying each in terms of four dimensions: allowed task differences, source tasks, transferred knowledge and learning methods. The methods are in order of appearance in the text and key to abbreviations are in Table 3.2.	57
3.2	List of abbreviations used in Table 3.1.	58
6.1	State predicates used in navigation domain class, where $\mathbf{R}$ is a room marker, $\mathbf{C}$ is a corridor marker, $\mathbf{M}$ is either a room or corridor marker, $\mathbf{D}$ is a door and $\mathbf{X}$ is any object.	91
6.2	Set of action predicates P_A used in navigation domain class, where $\mathbf{R}$ is a room marker, $\mathbf{C}$ is a corridor marker, $\mathbf{M}$ is either a room or corridor marker and $\mathbf{D}$ is a door.	92

LIST OF ALGORITHMS

1	Policy Iteration	38
2	ϵ -greedy(Q, s, ϵ)	39
3	First-visit MC	40
4	Monte Carlo ϵ -soft on-policy control	41
5	Q-Learning	42
6	Sarsa(λ)	43
7	Q-RRL	52
8	TILDE / TILDE-RT	53
9	AbsSarsa(λ) Active	63
10	AbsSarsa(λ) (Passive)	64
11	AbsProb-PI	70
12	AbsProb-RL	73
13	PRPL - algorithm to build a policy library	75
14	AbsProb-RL - multiple source tasks	79
15	ψ -reuse($Q, s, \pi_{ab}, \epsilon, \psi$)	84
16	S2L-RL - Simultaneous Two-layer Reinforcement Learning	88

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
CBR	Case-based reasoning
DP	Dynamic Programming
MC	Monte Carlo
MDP	Markov Decision Process
PI	Policy Iteration
POMDP	Partially Observable Markov Decision Process
PS	Policy search
RL	Reinforcement Learning
RRL	Relational Reinforcement Learning
RMDP	Relational Markov Decision Process
S2L-RL	Simultaneous Two-layer Reinforcement Learning
TL	Transfer Learning
<i>w.p.</i>	With probability

LIST OF SYMBOLS

α	Abstract action
α_t	Abstract action executed at time step t
γ	Discount-rate parameter
δ_W	Similarity factor between policies
δ_π	Step size function in AbsProb-PI/AbsProb-RL
ϵ	Probability of random action in ϵ -greedy strategy
$\eta(s, a)$	Eligibility trace for state-action pair s, a
$\eta_{ab}(\sigma, \alpha)$	Eligibility trace for abstract state-action pair σ, α
θ	A substitution of variables
λ	Decay-rate parameter for eligibility traces
μ	Learning rate
$\mu_G(i)$	Learning rate function for AbsProb-RL, depending on iteration i
ν	Decay-rate parameter of ψ in ψ -reuse exploration strategy
π	Policy
$\pi(s)$	Action taken in state s under deterministic policy π
$\pi(s, a)$	Probability of taking action a in state s under stochastic policy π
ϕ_a	Abstraction function for actions; maps ground actions into abstract actions
ϕ_s	Abstraction function for states; maps ground states into abstract states
σ	Abstract state
σ_t	Abstract state observed at time step t
Σ	Relational alphabet
τ	Temperature parameter in softmax function

ψ	Probability of reusing a past policy in ψ -reuse exploration strategy
Ω	Task
a	Ground action
a_t	Ground action executed at time step t
$\mathcal{A}$	Set of all ground actions
$\mathcal{A}(s)$	Set of feasible ground actions in state s
$\mathcal{A}_\alpha$	Set of all ground actions covered by abstract action α
$\mathcal{A}_\alpha(s)$	Set of all feasible ground actions covered by abstract action α in state s
$\mathcal{A}_{ab}$	Set of all abstract actions
b^0	Probability distribution for the initial state in a Markov Decision Process
$\mathbf{b}^0$	Matrix-vector representation of b^0
$\mathcal{B}$	Background knowledge
$\mathcal{D}$	Domain
$\mathcal{D}_C$	Domain class
$\mathcal{G}$	Set of goal states
G	Policy gradient
G_{local}	Local estimate of policy gradient in AbsProb-RL
G_{global}	Global estimate of policy gradient in AbsProb-RL
$H(i)$	Estimation window size function in AbsProb-RL, depending on iteration i
HB_L	Herbrand Base of a language L
HU_L	Herbrand Universe of a language L
HI_L	The set of every Herbrand Interpretation in a language L
$\mathcal{O}$	Set of observations
$P(X)$	Probability of X

Q	Action-value function
Q_{ab}	Abstract action-value function
r_t	Reward received at time step t
R	Reward function of a Markov Decision Process
$\mathbf{R}$	Matrix-vector representation of R
s	Ground state
s_0	Initial ground state
s_t	Ground state observed at time step t
$\mathcal{S}$	Set of ground states
$\mathcal{S}_\sigma$	Set of ground states covered by an abstract state σ
$\mathcal{S}_{ab}$	Set of abstract states
t	Discrete time step
t_{max}	Maximum number of steps within an episode
T	Transition function of a Markov Decision Process
$T^\pi(s, s')$	Probability of agent transitioning from state s to state s' when following policy π
$\mathbf{T}^\pi$	Matrix-vector representation of T^π
$V^\pi(s)$	Value function of policy π that gives the expected discounted sum of rewards for executing π starting from state s
$\mathbf{V}^\pi$	Matrix-vector representation of V^π
W^π	Expected discounted sum of rewards the agent receives when exploiting policy π , i.e., the value of policy π
$\mathcal{Z}(s, o)$	Probability of perceiving observation o in state s .

TABLE OF CONTENTS

1	INTRODUCTION	25
1.1	Objective	27
1.2	Contribution	27
1.3	Organization	28
2	BACKGROUND	29
2.1	Markov Decision Process	29
2.2	Value functions	31
2.3	Relational Markov Decision Process	32
2.4	State and Action Abstractions	34
2.5	Abstract policies	36
2.6	Planning	37
2.7	Reinforcement learning	38
2.7.1	Monte Carlo methods	39
2.7.2	Q-Learning	41
2.7.3	Sarsa(λ)	42
3	TRANSFER LEARNING PROBLEM AND RELATED WORK	45
3.1	Problem definition	45
3.2	Related work	47
3.2.1	Transfer methods with same state and action spaces	49
3.2.2	Hierarchical learning	50
3.2.3	Relational representation	51
3.2.4	Inter-task mappings	55
3.2.5	Multiple source tasks	56
3.3	Discussion	59
4	BUILDING ABSTRACT POLICIES	61
4.1	Building abstract policies from action-value function	61
4.1.1	AbsSarsa(λ) algorithm	62
4.1.2	Shortcomings of AbsSarsa(λ)	65
4.2	Building abstract policies using policy search	66
4.2.1	Stochastic abstract policies	67

4.2.2	AbsProb-PI algorithm	68
4.2.3	AbsProb-RL algorithm	69
4.3	Multiple source tasks	73
4.3.1	Building a policy library	74
4.3.2	Building a generalized policy	75
4.3.3	Illustrative example	76
5	LEARNING AGENT ARCHITECTURE	81
5.1	Overview	81
5.2	ψ -reuse exploration strategy	83
5.3	Simultaneous Two-layer Reinforcement Learning	84
6	EXPERIMENTS AND RESULTS	89
6.1	Navigation Domain Class	89
6.2	Experimental Setup	94
6.3	Parameter analysis in AbsProb-RL	94
6.4	S2L-RL with AbsSarsa(λ)	98
6.5	S2L-RL with AbsProb-RL	101
7	CONCLUSION AND FUTURE WORK	109
	REFERENCES	113

1 INTRODUCTION

There are many sequential decision problems that an autonomous agent faces for which a solution cannot be found in advance, because the agent may not have all the information needed to model them. *Reinforcement learning (RL)* is a very powerful framework to tackle such scenarios, as in autonomous robot navigation tasks (NAVARRO-GUERRERO et al., 2012; CONN; PETERS, 2007). In RL an agent repeatedly observes the state of its environment and selects actions. Performing an action changes the state of the world, and the agent also obtains an immediate numeric payoff as a result. The agent must learn to select actions (must learn a policy) so as to maximize a long-term sum or average of the future payoffs it will receive.

One issue with RL is that the agent may take a long time to learn appropriate behavior, as it is based on repetitive interactions of the robot with the environment by trial-and-error. In the conventional RL framework, the agent does not initially know what effects its actions have on the state of the world, nor what immediate payoffs its actions will produce. In particular, the agent does not know which action is the best to select at any given state. Rather, it must try out various actions in various states, and must gradually learn which action is best at each state so as to maximize the long run payoff. Alas, this whole process can be very time-consuming.

Hopefully an agent can improve its learning abilities if solutions for similar past problems can be used in the current problem. Besides composing their direct interaction in the current environment, agents can also benefit from their own past experiences, i.e., the knowledge acquired by the agent from solving one or more previous *source tasks* may be used to solve a new *target task* more effectively. This process of transferring knowledge is called *transfer learning* (TL) (PAN; YANG, 2010). Recent researches show that leveraging knowledge from similar tasks can improve an agent’s learning performance on a new task, by giving it a better starting point or accelerating learning, making the agent to show better behavior especially in the early stages of the learning process.

Indeed, much research in RL has increased learning speed by exploiting transfer learning. When designing a TL method, there are some important points to be considered, such as what type of knowledge is transferred and how the problem is represented. Regarding transferred knowledge, most methods provide the transfer of value functions (ASADA et al., 1996; TAYLOR; STONE; LIU, 2007), policies (TAYLOR; WHITESON; STONE, 2007) or partial policies (solutions to subtasks) (MEHTA et al., 2008; KONIDARIS; SCHEIDWASSER; BARTO, 2012). Besides, while most methods just take into ac-

count knowledge from only one source task, there are methods that transfer a policy library (FERNÁNDEZ; GARCÍA; VELOSO, 2010), containing several policies, one for each source task; or a generalized policy (SILVA; PEREIRA; COSTA, 2012), a policy that extracts common solutions for a set of source tasks.

As for the representation, in general the models are simple and with little semantics (e.g. enumerated states), being too attached to each problem making it hard to transfer knowledge across tasks. An alternative is to use a factored state representation, which allows agents to share experiences among similar states (BOUTILIER; DEARDEN; GOLDSZMIDT, 2000; VAN OTTERLO, 2004). Particularly, we can use a *relational representation* to represent factored states, which is a richer and generalized representation (MORALES, 2003). It uses objects and their relationships to describe the world, providing a natural abstraction over objects, thus enabling *generalization* of states and actions (and therefore, policies). This abstraction facilitates the transfer of knowledge between similar tasks, which do not need to be in the same domain; they only need to share features at an abstract level.

Given this abstraction scenario, some algorithms learn an abstract value function which accelerates learning, but may deteriorate policy quality (PAZIS; PARR, 2011). Because under abstraction the Bellman’s principle of optimality (BELLMAN, 1957) may not be observed, search directly in the space of abstract policies has played a key role to obtain optimal abstract policies (STULP; SIGAUD, 2012; DEISENROTH; RASMUSSEN, 2011; CASTRO; TAMAR; MANNOR, 2012). Abstract policies allow transfer of knowledge among different problems with the same factored representation.

Our work is based on the fact that many domains can be described in terms of objects and the relations among them. Tasks in different domains can be represented in a similar relational form, leading to direct and elegant way to abstract knowledge through the use of logical variables. In doing so, ground states are grouped into abstract states by the use of variables, allowing us to define abstract policies to represent knowledge about the solution of the concrete problem. Once an abstract policy is produced, it can be used in other learning problems. We explore the intuition that generalization from closely related, but solved, problems can produce policies that make good decisions in many states of a new unsolved problem.

In this dissertation we are interested in the following general questions: how can an agent transfer and reuse the past abstract knowledge from one or more source tasks to accelerate the learning process in future different tasks? Moreover, how can an agent generate such abstract knowledge while interacting with the environment, without having a model of the problem?

1.1 Objective

The goal of this work is to propose techniques for transfer learning in reinforcement learning. Specifically, these techniques should allow an abstract representation of knowledge, so as to generalize solutions across different but related tasks.

In particular, we consider the use of a relational representation not only for abstraction, but also to transfer knowledge between tasks with different states and actions. As abstract policies show a better generalization power than value functions, they become more appropriate for abstraction; hence such policies are most suitable for transfer learning. So, the idea here is that an agent autonomously learns an abstract policy from the source task and uses it to improve its learning performance on a target task. In addition, the agent gathers knowledge from multiple source tasks, not just one.

Thus, our goals are: (i) to present algorithms for building abstract policies described by relational representations, allowing the generalization of knowledge about the solution of one or more source tasks; (ii) to introduce an architecture where these abstract policies are used to affect the learning process of a target task.

1.2 Contribution

The main contributions of this work are two algorithms for building an abstract policy by solving RMDPs, AbsSarsa(λ) and AbsProb-RL, and an architecture that enables the use of these two algorithms, named Simultaneous Two-layer Reinforcement Learning agent architecture (S2L-RL).

The algorithms are both model-free, since they do not require a complete model of the problem, and work online, i.e., they learn while interacting with the environment. AbsSarsa(λ) uses a temporal-difference method based on Sarsa(λ) to find an optimal abstract value function from which a deterministic abstract policy is derived. AbsProb-RL uses direct search in the policy space to build a stochastic abstract policy. AbsProb-RL also deals with multiple source tasks, finding a single generalized abstract policy for all tasks.

The proposed S2L-RL architecture provides a learning framework for a reinforcement learning agent. This architecture has two levels of representation, abstract level and ground level, and combines the use of past abstract policies from the source tasks with simultaneous learning of two new policies: an abstract policy and a ground policy. These two policies represent the solution to the current problem and they are refined through exploration and interaction of the agent with the environment. AbsSarsa(λ) and AbsProb-RL are the algorithms used at the abstract level for building the abstract policy, while usual RL algorithms are used at the ground level. The abstract level generalizes knowledge

learned in the ground level; since the abstract space is smaller than the ground space, it allows faster learning in the abstract level. The knowledge learned in the abstract level is then fed back into the ground level, directing the search for an optimal solution; additionally, the abstract level builds an abstract policy that can be transferred to a number of similar problems.

We conduct experiments with an autonomous robotic navigation domain to evaluate the method effectiveness and the agent succeeds in leveraging past knowledge to present a faster learning and better behavior on a target task. We also compare its performance against another TL with policy reuse algorithm.

1.3 Organization

This document is organized as follows.

- **Chapter 2 – Background** presents concepts necessary to the full understanding of the work, such as Markov Decision Process and reinforcement learning;
- **Chapter 3 – Transfer learning** problem and related work: details the problem we want to solve and reviews works related to TL in reinforcement learning;
- **Chapter 4 – Building abstract policies** presents two algorithms, AbsSarsa(λ) and AbsProb-RL, proposed here for building abstract policies;
- **Chapter 5 – Learning agent architecture** describes the proposed architecture for the learning agent, called Simultaneous Two-layer Reinforcement Learning (S2L-RL), which enables transfer learning and uses abstraction;
- **Chapter 6 – Experiments** describes the navigation domain in which the experiments were carried out and presents the results;
- **Chapter 7 – Conclusions** summarizes main contributions and future work.

2 BACKGROUND

This chapter provides background information that lays the basis for the remainder of the dissertation. Section 2.1 reviews the model we use to describe the learning problems, the Markov Decision Process (MDP). Section 2.2 reviews value functions, fundamental elements in reinforcement learning. Section 2.3 explains Relational Markov Decision Process, a variation of MDPs with a relational description. Section 2.4 explains how a relational representation can be used in the abstraction of states and actions. Section 2.5 defines abstract policies and shows how they can be used. Section 2.6 reviews planning, a class of methods for solving MDPs. Finally, Section 2.7 reviews reinforcement learning.

2.1 Markov Decision Process

We are interested in sequential decision-making problems, i.e., the problem is fully solved by a sequence of actions. A formalism widely used for this kind of problem is the Markov Decision Process (MDP) (PUTERMAN, 1994). Its key concept is the Markov property: every state encodes all the information needed for taking the optimal decision in that state. We define the MDP as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \mathcal{G}, b^0 \rangle$, where:

- $\mathcal{S}$ is the set of *states*;
- $\mathcal{A}$ is the set of *actions*; additionally, we denote $\mathcal{A}(s)$ the set of feasible actions at state $s \in \mathcal{S}$;
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is a *transition function* such that $T(s, a, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability of reaching state s' at time $t + 1$ when the agent is in state s and executes action a at time t ;
- $R : \mathcal{S} \rightarrow \mathbb{R}$ is a bounded *reward function*, such that $r_t = R(s_t)$ is the reward received when the agent is in state s at time t ;
- $\mathcal{G} \subset \mathcal{S}$ is a set of goal states. No transition leaves any goal state, i.e., $T(s, a, s') = 0, \forall s \in \mathcal{G}, \forall a \in \mathcal{A}, \forall s' \in \mathcal{S}$;
- $b^0 : \mathcal{S} \rightarrow [0, 1]$ is the initial state probability distribution, such that $b^0(s)$ is the probability of state s being the first one in an episode, i.e., $P(s_0 = s) = b^0(s)$.

The agent stays in a constant perception-action cycle (Figure 2.1): at each time step t , it observes state $s_t \in \mathcal{S}$ and receives reward r_t ; then it chooses an action $a_t \in \mathcal{A}(s)$,

Figure 2.1 – *The perception-action cycle* – Agent observes current state from environment, choose an action to execute, receives reward and observes new state.

which leads it to another state s_{t+1} according to probabilities of T and then the cycle restarts. Here we deal with a type of MDP called *episodic* (SUTTON; BARTO, 1998), i.e., every time a goal state $s \in \mathcal{G}$ is reached an episode ends and a new episode starts in some initial state chosen according to b^0 .

Note that the environment can be non-deterministic; that is, taking the same action in the same state on two different occasions may result in two different next states each occasion. Besides, we assume the environment is stationary; this means that the probabilities of making state transitions or the reward values do not change over time.

Finding a solution to an MDP means finding a *policy* π that specifies which actions $a \in \mathcal{A}$ should be executed when the agent is in each state $s \in \mathcal{S}$. If the policy is deterministic, it is defined as $\pi : \mathcal{S} \rightarrow \mathcal{A}$, i.e., for each state $s \in \mathcal{S}$ the policy only points to one action $a \in \mathcal{A}$. In the general case, the policy can be non-deterministic and we define a stochastic policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, in which every action $a \in \mathcal{A}$ has an associated probability of being executed in each state $s \in \mathcal{S}$. An *optimal policy* π^* is a policy that maximizes some function R_t of the future rewards $r_t, r_{t+1}, r_{t+2}, \dots$. A common definition, which we use, is to maximize the sum of *discounted rewards* over an *infinite horizon*: $R_t = \sum_{i=t}^{\infty} \gamma^i r_i$, where $0 \leq \gamma < 1$ is the *discount factor*. It is known that in an MDP, the set of deterministic and memoryless policies $\pi : \mathcal{S} \rightarrow \mathcal{A}$ contains an optimal policy (LITTMAN, 1994; PUTERMAN, 1994). ‘Memoryless’ means that the policies do not use the history (sequence of visited states) or any other information but the current state to indicate an action to take. The Markov property, explained previously, assures that the current state is enough for the optimal decision.

In this work we also assume the agent has *total observability* of the problem, that is, it always can sense in which state it is and, therefore, according to the Markov condition, the

optimal policy only depends on the current state. However, there are many cases in which the environment is only *partially observable*, as a result of which the agent does not know for certain in which state it actually is. This kind of problems are defined as POMDPs (Partially Observable Markov Decision Processes) (SILVER; VENESS, 2010). POMDPs are defined the same way MDPs are, with the addition of an observation set $\mathcal{O}$ and observation probabilities $\mathcal{Z}(s, o)$ that indicate the probability of perceiving observation $o \in \mathcal{O}$ in state $s \in \mathcal{S}$. Thereby, when the agent makes an observation, instead of establishing which is its current state, it establishes a *belief state*, which is a probability distribution among all possible states. In its interaction with the environment, the agent perceives an observation o_t , updates its belief state with the probabilities of actually being in state s_1, s_2 , etc., executes an action a_t (the transition $T(s_t, a_t, s_{t+1})$ occurs with the function T intrinsic to the problem, even if the agent fails to notice s_t and s_{t+1}) and then a reward r_t and a new observation o_{t+1} are perceived by the agent. For complex POMDPs, it is very hard to find approximate optimal policies; in fact, it is a PSPACE-hard problem (RUSSELL; NORVIG, 2003).

2.2 Value functions

Many algorithms to find a policy for an MDP are based on estimating *value functions* (SUTTON; BARTO, 1998). These are functions that estimate the expected future rewards when the agent is at each state, i.e., the higher the value of the state, the better in terms of expected return. Value functions are defined for a particular policy, as future rewards depend on what actions will be taken. The *state-value function* for policy π , $V^\pi : \mathcal{S} \rightarrow \mathbb{R}$, is defined as:

$$V^\pi(s) = \mathbf{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t | \pi, s_0 = s \right], \forall s \in \mathcal{S}, \quad (2.1)$$

where $V^\pi(s)$ is the expected return when starting in state s and following policy π thereafter.

Optimal policy π^* is the policy which satisfies the condition $V^{\pi^*}(s) \geq V^\pi(s), \forall s \in \mathcal{S}, \forall \pi$. The state-value function for an optimal policy is shortly denoted V^* , so:

$$V^*(s) = \max_{\pi} V^\pi(s), \forall s \in \mathcal{S}. \quad (2.2)$$

Therefore, the interest lies in finding a policy that is the closest or, ideally, equal to the optimal policy π^* .

Similarly, we define a more specific value function considering state-action pairs. The value of taking action a in state s under a policy π is denoted as $Q^\pi(s, a)$, i.e, it is the

expected return starting from state s , taking action a and thereafter following policy π . $Q^\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is called the *action-value function* for policy π and it is defined as:

$$Q^\pi(s, a) = \mathbf{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t | \pi, s_0 = s, a_0 = a \right], \forall s \in \mathcal{S}. \quad (2.3)$$

We define the optimal action-value function Q^* , which is shared among optimal policies, as:

$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}. \quad (2.4)$$

We also define the *value of a policy* (NG; JORDAN, 2000) with respect to initial state distribution b^0 . W^π , the value of a policy π , is defined as:

$$W^\pi = \mathbf{E}_{s_0 \sim b^0} [V^\pi(s_0)] \quad (2.5)$$

$$= \sum_{s \in \mathcal{S}} b_0(s) [V^\pi(s)] \quad (2.6)$$

$$= \sum_{s \in \mathcal{S}} b_0(s) \left\{ \mathbf{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t | \pi, s_0 = s \right] \right\} \quad (2.7)$$

where the subscript $s_0 \sim b^0$ indicates that the expectation is with respect to s_0 drawn according to b^0 .

The simplest representation for states and actions is to enumerate them all (e.g. states are simply numbered $1, 2, 3, \dots$). However, enabling transfer with this representation requires a mapping function between the source and target tasks. It is possible to avoid the mapping problem by ensuring that the source and target tasks have the same representation (TORREY; SHAVLIK, 2009). If states, actions, value functions and policies of the source task are represented using a language identical of the target task, they can be transferred directly with no translation. Next section describes a representation that can be shared among tasks.

2.3 Relational Markov Decision Process

A Relational MDP (RMDP) (VAN OTTERLO, 2009) is an extension of the MDP formalism by using a relational alphabet to describe states and actions (and thus, policies). A relational representation is useful for creating domains that naturally produce common task representations and its use facilitates the generalization of the problem and the solution (DZEROSKI; DE RAEDT; DRIESSENS, 2001), which in turn facilitates knowledge transfer among tasks.

First, we introduce some definitions. A relational alphabet Σ is a set of constants and predicates. If $\mathbf{t}_1, \dots, \mathbf{t}_n$ are *terms*, each one being a constant (represented with lower-case

letters) or simply a variable (represented with capital letters), and if p/n is a predicate symbol p with arity $n \geq 0$, then $p(t_1, \dots, t_n)$ is an *atom*. If an atom does not contain any variable, it is called a *ground atom*. A set of atoms is a *conjunction* and it is a ground conjunction if it contains only ground atoms. Suppose a language L , which uses predicates and terms constructed from names in the signature $Sig(L) = \langle P, C \rangle$, where P is the set of predicates and C is the set of constants (here we consider there are no function symbols). The Herbrand Base of L , HB_L , is the set of ground atoms using constants of C in the predicates of P . The Herbrand Interpretations, HI_L , is the set of every interpretation of HB_L , i.e., every truth value combination for ground atoms in HB_L .

Then we define the RMDP as a tuple $\langle \Sigma, \mathcal{B}, \mathcal{S}, \mathcal{A}, T, R, \mathcal{G}, b^0 \rangle$, where:

- $\Sigma = C \cup P_S \cup P_A$ is a finite relational alphabet such that C is a set of *constants*, which represents the objects of the environment; P_S is a set of *state predicates* used to describe properties of and relations among objects; and P_A is a set of *action predicates*;
- $\mathcal{B}$ is the *background theory*, a set of clauses over L which defines constraints for the combinations of predicates and objects;
- Given the language L_S with signature $Sig(L_S) = \langle P_S, C \rangle$, the state space $\mathcal{S}$ of the RMDP is defined as a subset of Herbrand Interpretations HI_{L_S} that satisfies the constraints imposed by $\mathcal{B}$, i.e., $\mathcal{S} \subset \{s \in HI_{L_S} | s \models \mathcal{B}\}$;
- Action set $\mathcal{A}$ is a subset of HB_{L_A} , where L_A is a language with signature $Sig(L_A) = \langle P_A, C \rangle$, again satisfying $\mathcal{B}$;
- T , R , $\mathcal{G}$, and b^0 are defined as in an MDP.

We adopt a closed world assumption: if a ground atom does not appear in an interpretation, the negated ground atom is assumed, as shown in Example 1:

Example 1 If $P_S = \{p_1/1, p_2/1, p_3/1\}$ and $C = \{a, b\}$, then $p_1(a) \wedge p_2(b)$ denotes the state $s_1 = p_1(a) \wedge p_2(b) \wedge \neg p_1(b) \wedge \neg p_2(a) \wedge \neg p_3(a) \wedge \neg p_3(b)$.

At this point this closed world assumption is merely a notational simplification to compactly express maximal clauses; later the closed world assumption will be important during abstraction.

Simply put, an RMDP is an MDP in which the states and the actions are represented using a relational language. RMDPs describe the same class of problems as MDPs, there is not a single RMDP that cannot be described as an MDP and vice-versa. The same algorithms and methods used for MDPs are all valid for RMDPs. We explore the relational

representation mainly because it can provide abstraction due to the use of variables, which can be explored in reinforcement learning tasks. Furthermore, relational representations allow us to apply the results of learning in a simpler domain to learning in a more complex domain (DZEROSKI; DE RAEDT; DRIESSENS, 2001).

2.4 State and Action Abstractions

A state $s \in \mathcal{S}$ is called a *ground state* if it contains no variables. Similarly, an action $a \in \mathcal{A}$ is called a *ground action* for the same reason. A relational representation enables us to aggregate states and actions by using variables instead of constants in the predicate terms.

Example 2 Consider a state where a robot is in a room r_1 ; that is, the state contains $\text{inRoom}(r_1)$. Another state contains $\text{inRoom}(r_2)$. An abstract state that abstracts both states contains $\text{inRoom}(X)$, meaning that there exists a room such that the robot is in it.

A *substitution* θ is a set $\{X_1/t_1, \dots, X_n/t_n\}$, binding each variable X_i to a term t_i ; it can be applied to a term, atom or conjunction. We call an *abstract state* σ (and *abstract action* α) a conjunction with no ground atom, where each variable in a conjunction is implicitly assumed to be existentially quantified. An abstract state (action) is generated from a ground state (action) when constants in the ground state (action) are replaced by variables¹. That is, we only consider abstractions of the following form: given a ground state containing non-negated atom $p_1(a)$, replace this atom by $p_1(X)$ where X is a fresh logical variable (and do it for all non-negated ground atoms of the ground state) — the resulting expression represents an abstract state that abstracts the ground state.

We denote by $\mathcal{S}_\sigma$ the set of ground states covered by an abstract state σ . Similarly, we define $\mathcal{A}_\alpha(s)$ as the set of all ground actions covered by an abstract action α in ground state s . We also define $\mathcal{S}_{ab}$ and $\mathcal{A}_{ab}$ as the set of all abstract states and the set of all abstract actions in an RMDP, respectively. Examples 3 and 4 illustrate abstract state and actions.

Example 3 Consider an abstract state $\sigma = \{p_1(X_1), p_2(X_1, X_2)\}$, $\sigma \in \mathcal{S}_{ab}$; a ground state $s_1 \in \mathcal{S}$, $s_1 = \{p_1(t_1), p_2(t_1, t_2)\}$ is abstracted by σ with $\theta = \{X_1/t_1, X_2/t_2\}$ and a ground state $s_2 \in \mathcal{S}$, $s_2 = \{p_1(t_3), p_2(t_3, t_4)\}$ is abstracted by σ with $\theta = \{X_1/t_3, X_2/t_4\}$. In this case, σ represents an abstraction of both s_1 and s_2 .

¹ Replacing just one or more constants with variables would also achieve a certain level of abstraction. However, in this work we consider only the level of abstraction that preserves every predicate in the conjunction without any constants in the terms.

Example 4 Now, consider a background theory $\mathcal{B} = \{\mathbf{a}_1(\mathbf{X}) \Rightarrow \mathbf{p}_1(\mathbf{X})\}$ and the abstract action $\alpha = \mathbf{a}_1(\mathbf{X})$. In this case, if the agent is in ground state $s = \{\mathbf{p}_1(\mathbf{t}_1), \mathbf{p}_2(\mathbf{t}_1, \mathbf{t}_2)\}$, the abstract action α covers ground action $a = \mathbf{a}_1(\mathbf{t}_1)$ under substitution $\theta = \{\mathbf{X}/\mathbf{t}_1\}$, but it does not cover ground action $a' = \mathbf{a}_1(\mathbf{t}_2)$. On the other hand, if the agent is in ground state $s' = \{\mathbf{p}_1(\mathbf{t}_2), \mathbf{p}_2(\mathbf{t}_1, \mathbf{t}_2)\}$, abstract action α does not cover ground action a , but it does cover ground action a' under substitution $\theta' = \{\mathbf{X}/\mathbf{t}_2\}$.

Hence each σ represents an aggregate of ground states, and the abstract state space partitions the original ground state space $\mathcal{S}$ into a set of k subsets $\mathcal{S}_{\sigma_1}, \dots, \mathcal{S}_{\sigma_k}$, where $k = |\mathcal{S}_{ab}|$. That is, $\mathcal{S} = \cup_{i=1}^k \mathcal{S}_{\sigma_i}$ and $\mathcal{S}_{\sigma_i} \cap \mathcal{S}_{\sigma_j} = \emptyset$, $i \neq j$.

Li, Walsh and Littman (2006) propose an unified theory for state abstraction in MDPs. Here we summarize this theory. A state abstraction function is defined by $\phi_s : \mathcal{S} \rightarrow \mathcal{S}_{ab}$; $\phi_s(s)$ is the abstract state corresponding to ground state s . There are many abstractions for an MDP, but some are more useful because they preserve critical information for solving the original MDP. Given any states $s_1, s_2 \in \mathcal{S}$, five types of state abstraction are defined:

1. A model-irrelevance abstraction ϕ_s^{model} that aggregates states with same reward and transitions, i.e., $R(s_1) = R(s_2)$ and $T(s_1, a, s') = T(s_2, a, s'), \forall a \in \mathcal{A}, \forall s' \in \mathcal{S}$;
2. A Q^π -irrelevance abstraction $\phi_s^{Q^\pi}$ that aggregates states with same action-value for any policy, i.e., $\phi_s^{Q^\pi}(s_1) = \phi_s^{Q^\pi}(s_2) \Rightarrow Q^\pi(s_1, a) = Q^\pi(s_2, a), \forall \pi, a \in \mathcal{A}$;
3. A Q^* -irrelevance abstraction $\phi_s^{Q^*}$ that aggregates states with same optimal action-value, i.e., $\phi_s^{Q^*}(s_1) = \phi_s^{Q^*}(s_2) \Rightarrow Q^*(s_1, a) = Q^*(s_2, a), \forall \pi, a \in \mathcal{A}$;
4. An a^* -irrelevance abstraction $\phi_s^{a^*}$ that aggregates states with same action-value for the optimal action, i.e., $\phi_s^{a^*}(s_1) = \phi_s^{a^*}(s_2) \Rightarrow Q^*(s_1, a^*) = \max_a Q^*(s_1, a) = \max_a Q^*(s_2, a) = Q^*(s_2, a^*)$;
5. A π^* -irrelevance abstraction $\phi_s^{\pi^*}$ that aggregates states with same optimal policy, i.e., $\phi_s^{\pi^*}(s_1) = \phi_s^{\pi^*}(s_2) \Rightarrow Q^*(s_1, a^*) = \max_a Q^*(s_1, a)$ and $Q^*(s_2, a^*) = \max_a Q^*(s_2, a)$.

These five abstraction functions form a hierarchy and have a relation of partial order regarding granularity of the abstraction. The relation $\phi_s^1 \succeq \phi_s^2$ means that ϕ_s^1 is finer than ϕ_s^2 , i.e., ϕ_s^1 partitions the state space in more abstract states than ϕ_s^2 . The partial ordering for the five types is: $\phi_s^{model} \succeq \phi_s^{Q^\pi} \succeq \phi_s^{Q^*} \succeq \phi_s^{a^*} \succeq \phi_s^{\pi^*}$. In this work we focus on the coarser type of abstraction, $\phi_s^{\pi^*}$.

Having defined abstract states and abstract actions, we can now define how solutions to RMDPs can be represented in an abstract form. Next section presents abstract policies.

2.5 Abstract policies

A deterministic abstract policy specifies an abstract action for each abstract state: $\pi_{ab} : \mathcal{S}_{ab} \rightarrow \mathcal{A}_{ab}$. The challenge is to apply an abstract policy in a ground problem: we must provide a way to translate from ground to abstract level, and vice-versa. For this purpose, we define two operations: *abstraction* and *grounding*. First, let us focus on deterministic policies.

Abstraction is the translation from the ground level to the abstract level. We have already defined an abstraction function for states $\phi_s : \mathcal{S} \rightarrow \mathcal{S}_{ab}$, which maps ground states to abstract states. For a ground state s , the corresponding abstract state σ is given by $\phi_s(s) = \sigma$ so that $s \in \mathcal{S}_\sigma$. Similarly, we define an abstraction function for actions $\phi_a : \mathcal{A} \rightarrow \mathcal{A}_{ab}$, which maps ground actions to abstract actions. For a ground action a , the corresponding abstract action α is given by $\phi_a(a) = \alpha$ so that $a \in \mathcal{A}_\alpha$.

Grounding is the translation from the abstract level to the ground level. Given an abstract state σ , the procedure $grounding(\sigma)$ returns one state s from the set of possible states $\mathcal{S}_\sigma$; note that an abstract state may be mapped into a set of ground states for the underlying ground decision problem. The selected state is selected randomly with a uniform distribution. This process is denoted by $s = grounding(\sigma)$. To produce a ground action from an abstract action, we need also a ground state s because the ground actions available depend on the state. $grounding(\alpha, s)$ returns a ground action a from the set of possible actions $\mathcal{A}_\alpha(s)$. Again, the selected action is selected randomly with a uniform distribution. This process is denoted by $a = grounding(\alpha, s)$ ².

We then propose the following scheme to apply an abstract policy in a ground problem. Assume a deterministic abstract policy π_{ab} is given. Agent observes ground state s and we find the corresponding abstract state $\sigma = \phi_s(s)$. Abstract policy $\pi_{ab}(\sigma)$ defines an abstract action α to be applied in the abstract state σ . To produce a particular ground action a , we apply the grounding operation over α and s , i.e., $a = grounding(\pi_{ab}(\sigma), s)$.

For stochastic abstract policies, the process for applying it to a ground problem is similar. If the policy is stochastic, then it is defined as $\pi_{ab} : \mathcal{S}_{ab} \times \mathcal{A}_{ab} \rightarrow [0, 1]$. For each pair of abstract state and abstract action $(\sigma_i, \alpha_j) \in \mathcal{S}_{ab} \times \mathcal{A}_{ab}$, $\pi_{ab}(\sigma_i, \alpha_j)$ points out a probability $P(\alpha_j | \sigma_i)$. After finding abstract state $\sigma = \phi_s(s)$ from observed ground state s , abstract policy points probabilities $\pi_{ab}(\sigma, \alpha_k) = P(\alpha_k | \sigma)$ for all $\alpha_k \in \mathcal{A}_{ab}$. We select an abstract action $\alpha_k \in \mathcal{A}_{ab}$ according to these probabilities. Then the process remains the same, with $a = grounding(\alpha_k, s)$.

² In this work we actually never use the grounding function for abstract states. Here we are only concerned in using abstract policies, that take a ground state and choose an abstract action. Therefore, we use only the grounding of abstract actions.

The following sections show the main methods and algorithms for finding optimal (ground) policies for a given (R)MDP. The concepts in them are also of great importance for finding optimal abstract policies. Algorithms specific for building abstract policies are found in Chapters 3 and 4.

2.6 Planning

Given an (R)MDP, one is interested in finding the optimal policy. One possible way to do this is to use dynamic programming (DP) algorithms. DP refers to a collection of algorithms that compute optimal policies given a perfect model of the environment as an MDP. DP is related directly to the *planning* field in AI; planning refers to solving problems by making a plan of action given a model and when planning is complete the action is taken (SUTTON, 1991).

Policy Iteration (PI) (HOWARD, 1960) is a DP algorithm based on the fact that the optimal state-value function V^* leads to the optimal policy. It has repetitive iterations of two steps: policy evaluation and policy improvement. The idea is to start with any initial policy π_0 and iteratively improve it until convergence to π^* . At each iteration, first the state-value function is calculated (policy evaluation):

$$\begin{aligned} V^\pi(s) &= \mathbf{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t | \pi, s_0 = s \right] \\ &= R(s) + \gamma \sum_{s' \in \mathcal{S}} T(s, \pi(s), s') V^\pi(s') \forall s \in \mathcal{S}. \end{aligned} \quad (2.8)$$

Then, having determined the value function V^π , we do a policy improvement step. This step changes the current policy to a greedy policy that takes the best action with one step of look-ahead. This change give us a strictly better policy than the older, or at least a policy that is as good as, in which case the policy is already optimal. The policy improvement step is given by

$$\pi'(s) = \arg \max_{a \in \mathcal{A}} R(s) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') V^\pi(s') \forall s \in \mathcal{S}. \quad (2.9)$$

Algorithm 1 presents the complete algorithm.

Dynamic Programming algorithms are inside a class of techniques called *model-based*, because they require the entire (R)MDP model in order to work. However, there are many situations the entire model is not available for the agent. For these cases, we use *model-free* algorithms, i.e, algorithms that do not require the knowledge of the entire model beforehand. Reinforcement learning (RL) has a large class of model-free algorithms and they can be used to find an optimal memoryless policy for an (R)MDP.

Algorithm 1 Policy Iteration

```

▷ Given discount factor  $\gamma$  and a small positive number  $\epsilon_V$ 
1: Initialize  $V(s)$  and  $\pi(s)$  arbitrarily for all  $s \in \mathcal{S}$ 
2: repeat
3:   repeat ▷ Policy Evaluation
4:      $\Delta_V \leftarrow 0$ 
5:     for each  $s \in \mathcal{S}$  do
6:        $v \leftarrow V(s)$ 
7:        $V(s) \leftarrow R(s) + \gamma \sum_{s' \in \mathcal{S}} T(s, \pi(s), s') V(s')$ 
8:        $\Delta_V \leftarrow \max(\Delta_V, |v - V(s)|)$ 
9:   until  $\Delta_V < \epsilon_V$  ▷ Policy Improvement
10:   $policy\_stable \leftarrow true$ 
11:  for each  $s \in \mathcal{S}$  do
12:     $b \leftarrow \pi(s)$ 
13:     $\pi(s) \leftarrow \arg \max_a R(s) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') V(s')$ 
14:    if  $b \neq \pi(s)$  then
15:       $policy\_stable \leftarrow false$ 
16: until  $policy\_stable$ 
17: return  $\pi$ 

```

2.7 Reinforcement learning

In RL, the agent uses its own experiences with the environment to learn the policy. The agent learns at the same time it interacts with the environment. For this reason, it is also called *online* learning, whereas planning algorithms are referred as *offline* learning. The agent uses the numeric rewards (the reinforcement) received during its experiences and tries to maximize them by trial-and-error. In most interesting cases, actions not only affect immediate rewards but also future states and, consequently, future rewards. These two characteristics: trial-and-error and delayed reward are the most important for reinforcement learning (SUTTON; BARTO, 1998).

Here we focus on model-free RL methods, in which the agent does not need to know the entire (R)MDP model to perform learning. Most notably, the transition function T and the reward function R are unknown.

One challenge that arises with RL is the need to balance between exploitation and exploration. As the agent learns during execution, it maintains estimates of the action values and refines them during the learning process. If the agent chooses the action with the greatest value at that time, it is acting *greedily*, i.e., it is *exploiting* its current knowledge. On the other hand, if it chooses any other action, it is *exploring*, because this

enables it to improve the action estimates and thus discover other sequences of actions that are actually better. Exploitation may give the agent the maximum reward for that one action, but exploration may result in a greater reward in the long run (KAELBLING; LITTMAN; MOORE, 1996). Exploration is necessary for the convergence to the optimal policy and that is why balancing it with exploitation is important.

One simple, yet effective and popular strategy for balancing is the ϵ -greedy strategy. This method exploits most of the time and explores every once in a while. This exploration occurs with a small probability ϵ , when the agent selects an action at random, uniformly. The greedy action is selected with probability $1 - \epsilon$. It is detailed in Algorithm 2.

Algorithm 2 ϵ -greedy(Q, s, ϵ)

▷ Given the action-value function Q , current state s and exploration rate parameter ϵ

1: **if** $\text{random}([0, 1]) \leq \epsilon$ **then** ▷ uniform distribution over $[0, 1]$

2:      $a \leftarrow \text{random } a \in \mathcal{A}(s)$ ▷ Exploration

3: **else**

4:      $a \leftarrow \max_{a'} Q(s, a')$ ▷ Exploitation

5: **return** a

Two fundamental classes of model-free methods for solving reinforcement learning problems are *Monte Carlo* (MC) methods and *temporal-difference* (TD) learning. First we discuss MC methods in Section 2.7.1. Then we show two TD learning methods: Q -Learning, in Section 2.7.2 and Sarsa(λ), in Section 2.7.3.

2.7.1 Monte Carlo methods

Monte Carlo methods (METROPOLIS; ULAM, 1949) are a broad class of algorithms that rely on repeated random sampling to obtain numerical results. They are used in a number of areas and can also be used for solving RL problems based on averaging sample returns. They require only experience – sample sequences of states, actions and rewards – from online interaction with the environment, thus not requiring the complete model. A MC method involves deliberate use of random numbers in a calculation that has the structure of a stochastic process (KALOS; WHITLOCK, 2008). It is named after the Monte Carlo casino located in Monaco, due to its significant random component.

A Monte Carlo method can be used to learn the state-value function V^π for a policy π . Recall that the value of a state is the expected return starting from that state. While Policy Iteration calculates it from the model, MC can simply estimate it from experience, averaging returns observed after visits to that state. The average should converge to the expected value as the number of observed returns increase. This idea underlies all Monte

Carlo methods (SUTTON; BARTO, 1998). Having estimated V^π , this value could be used for making a policy improvement step in PI, for example.

Given a set of episodes following policy π and passing through state s , we can estimate $V^\pi(s)$. Each occurrence of state s in an episode is called a visit to s . The *first-visit MC* method averages the return following first visits to s . By the law of large numbers, the sequence of averages converges to the expected value of V^π as the number of visits goes to infinity. First-visit MC is described in Algorithm 3.

Algorithm 3 First-visit MC

```

▷ Given policy  $\pi$  to be evaluated
1: Initialize state-value function  $V^\pi$  arbitrarily
2:  $Rewards(s) \leftarrow$  an empty list,  $\forall s \in \mathcal{S}$ 
3: repeat
4:   Generate an episode using  $\pi$ 
5:   for each state  $s$  appearing in the episode do
6:      $r \leftarrow$  total reward following the first occurrence of  $s$ 
7:     Append  $r$  to  $Rewards(s)$ 
8:      $V^\pi(s) \leftarrow average(Rewards(s))$ 
9: until some stop criterion is met
10: return  $V^\pi$ 

```

If we want to estimate an action-value function Q^π the ideas are the same as for estimating V^π with the addition of actions when accounting for visits, but there is a complication. If π is a deterministic policy, then following π will only produce observations of only one action per state, leaving many state-action pairs unvisited. An approach to assure that all state-action pairs are encountered is to consider only stochastic policies with nonzero probability for all actions. We can use ϵ -soft policies, in which all actions have a minimum probability of $\frac{\epsilon}{|\mathcal{A}(s)|}$ for all state $s \in \mathcal{S}$. When searching for an optimal policy, an ϵ -soft policy can at most move to an ϵ -greedy policy, in which the greedy action has the probability $1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$ and all others have $\frac{\epsilon}{|\mathcal{A}(s)|}$, for every state $s \in \mathcal{S}$.

We said that we could obtain an optimal policy π^* by alternating Monte Carlo policy evaluation steps (e.g. with first-visit MC), estimating V^π or Q^π , with policy improvement steps, as in Equation 2.9. However, first-visit MC assumes there will be an infinite number of episodes, which is not feasible in practice. An alternative to avoid the infinite number of episodes is to move toward Q^π every episode, but not expecting to actually get closer until many steps. After each episode, the observed returns are used for policy evaluation and then the policy is improved at all states visited in that episode.

The Monte Carlo methods presented learn value functions and optimal policies from sampled experiences. Compared to DP methods, there are two main differences. First,

Algorithm 4 Monte Carlo ϵ -soft on-policy control

▷ Given minimum probability ϵ
 1: Initialize $\forall s \in \mathcal{S}, a \in \mathcal{A}$:
 2:    $Q(s, a)$ arbitrarily
 3:    $\pi(s, a)$ arbitrarily such that $\pi(s, a) \geq \frac{\epsilon}{|\mathcal{A}(s)|}$ and $\sum_{a \in \mathcal{A}} \pi(s, a) = 1 \ \forall s \in \mathcal{S}$
 4:    $Rewards(s, a) \leftarrow$ an empty list
 5:
 6: **repeat**
 7:   Generate an episode using π
 8:   **for** each pair s, a appearing in the episode **do**
 9:      $r \leftarrow$ total reward following the first occurrence of s, a
 10:     Append r to $Rewards(s)$
 11:      $Q(s) \leftarrow average(Rewards(s, a))$
 12:   **for** each s in the episode **do** $a^* \leftarrow \arg \max_a Q(s, a)$
 13:     **for** each $a \in \mathcal{A}$ **do**
 14:        $\pi(s, a) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \epsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$
 15: **until** some stop criterion is met
 16: **return** Q, π

---

they do not need a complete model to operate, they are model-free. Second, they do not update their value estimates on the basis of the value estimates of successor states, i.e., they do not bootstrap. This is an advantage in situations when the Markov property is violated.

2.7.2 Q-Learning

Temporal-difference methods are a combination of Monte Carlo and Dynamic Programming. TD methods learn directly from experiences, like MC methods. TD methods also update estimates based in part on other learned estimates, bootstrapping like DP methods. Here we describe Q-Learning, a TD method.

Q-Learning (WATKINS, 1989) is one of the most famous algorithms in reinforcement learning, having been extended numerous times. The agent uses experiences to directly learn the optimal *action-value* function Q^* . It has basically just one iterative step, which updates the estimate of Q :

$$Q(s_t, a_t) \leftarrow (1 - \mu)Q(s_t, a_t) + \mu[r_t + \gamma \max_a Q(s_{t+1}, a)], \quad (2.10)$$

where s_t is the observed state at time step t and a_t is the executed action at time step t ; r_t is the received reward after executing a_t ; $\mu, 0 \leq \mu \leq 1$, is the learning rate, which determines to what extent newly acquired information overrides old information; and γ , $0 \leq \gamma < 1$, is the discount factor which determines the importance of future rewards. Given enough time, the Q estimate converges to Q^* . The complete algorithm is described in Algorithm 5.

The higher the $Q(s, a)$ value is, more reward is expected by executing action a in state s . Therefore, to build a policy based on Q , choosing the action with highest Q value for each state is the best choice. That is:

$$\pi(s) = \max_{a \in \mathcal{A}} Q(s, a), \forall s \in \mathcal{S}. \quad (2.11)$$

Algorithm 5 Q-Learning

▷ Given discount factor γ and learning rate μ
1: Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
2: Observe initial state s_0
3: $s \leftarrow s_0$
4: **repeat**
5:     Choose action a
6:     Execute a , receive reward r and observe new state s'
7:     Update Q : $Q(s, a) \leftarrow (1 - \mu)Q(s, a) + \mu[r + \gamma \max_{a'} Q(s', a')]$
8:      $s \leftarrow s'$
9: **until** some stop criterion is met
10: **return** Q

2.7.3 Sarsa(λ)

Sarsa(λ) (RUMMERY; NIRANJAN, 1994; SUTTON, 1996) is another conventional TD algorithm to find optimal policy π^* . The idea behind this algorithm is as follows. It also estimates Q function, like Q-Learning. At time step t the experience $\langle s_t, a_t, r_t, s_{t+1}, a_{t+1} \rangle$ is used to update the eligibility trace function η and the Q -value estimate Q , as shown in Algorithm 6, where $0 \leq \lambda \leq 1$ is the decay rate of the eligibility trace, $0 \leq \gamma < 1$ is the discount factor, and $0 \leq \mu \leq 1$ is the learning rate. The eligibility trace function starts identically zero, and in episodic tasks it is reinitialized to zero after every episode. At each step t , a policy based on Q is used, such as the greedy policy $\pi(s_t) = \arg \max_{a \in \mathcal{A}} Q(s_t, a)$, which assigns the best action a to the state s_t . Besides the greedy choice, there can also be exploration steps, using the ϵ -greedy strategy for example.

Algorithm 6 Sarsa(λ)

▷ Given discount factor γ , decay rate λ , learning rate μ and maximum number of steps per episode t_{max}
 1: Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
 2: **for** each episode **do**
 3:   (Re)initialize $\eta(s, a)$ with 0 for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
 4:   Observe state s
 5:   Choose action a using a policy derived from Q
 6:   **for** each episode step $t \in \{0, 1, \dots, t_{max}\}$ or until $s \in \mathcal{G}$ **do**
 7:     Take action a , observe new state s' and receive reward r
 8:     Choose next action a' using a policy derived from Q (e.g. ϵ -greedy)
 9:      $\delta_Q = r + \gamma Q(s', a') - Q(s, a)$
 10:      $\eta(s, a) \leftarrow \eta(s, a) + 1$
 11:     **for** all s_i in $\mathcal{S}$ and a_j in $\mathcal{A}$ **do**
 12:        $Q(s_i, a_j) \leftarrow Q(s_i, a_j) + \mu \eta(s_i, a_j) \delta_Q$
 13:        $\eta(s_i, a_j) \leftarrow \gamma \lambda \eta(s_i, a_j)$
 14:      $s \leftarrow s'$
 15:      $a \leftarrow a'$
 16: **return** Q

Then in order to build a deterministic memoryless policy from the learning algorithm, we just use the maximum Q-values for each state, i.e., the greedy policy again (see Eq. 2.11).

The Sarsa(λ) algorithm finds an optimal policy for a ground (R)MDP. Exploring the knowledge at an abstract level, using the abstraction provided by an RMDP, enabling generalization and transfer learning, is one of the objectives of this work. This knowledge can be in the form of abstract policies or abstract value functions, for example. Next chapter we show related work that achieve transfer learning or this use of abstraction in MDPs.

3 TRANSFER LEARNING PROBLEM AND RELATED WORK

In this chapter we describe in detail the transfer learning problem we want to tackle, what it has been done to solve it, the current gaps in these solutions and what we propose to fill them. In Section 3.1 we provide some definitions and restrictions to clearly define the problem and scope of this work. Then Section 3.2 describes works that are related to this problem.

3.1 Problem definition

We are interested in the general problem of *Reinforcement Learning*, i.e., given the interactions of an agent within an (R)MDP, the agent must pursue the optimal policy. On the previous chapter, we presented the definitions of an RMDP and reinforcement learning. Here we specify the exact kind of problems we are dealing with in this work. The (R)MDPs we want to solve in this work share the following characteristics:

- *Non-deterministic transition function T* : when the agent executes an action a in state s_1 , the outcome is not certain; it may transit to s_2 , but it may also transit to s_3 or even stay in s_1 . It all depends on the probabilities of T .
- *Stationary environment*: the dynamics of the environment do not change over time; transition function T and reward function R remain unchanged from the beginning until the end of the learning process.
- *Episodic*: tasks have a goal state and when it is reached, the task ends. An episode is the period of time between the moment the task begins, when the agent is at an initial state, and the moment it ends, when the agent reaches a goal state. Alternatively, an episode may also end if the agent reaches a time limit without reaching the goal.
- *Total observability*: agent has total observability of the current state, i.e., it knows exactly which state it is at.

Furthermore, we are only considering as solutions the class of *memoryless* policies, i.e., policies that choose actions only according to the current observation. As MDPs possess the Markov property, it is always possible to find a memoryless policy that is optimal (LITTMAN, 1994). An advantage of memoryless policies is that they are simpler, thus easier and faster to be applied. Besides, as we are concerned about transferring

knowledge, they generalize better than memory-based policies because they do not need to keep track of the history.

We are interested in the problem of RL when the agent can exploit knowledge obtained from similar previous tasks. In order to define which tasks are allowed, we present some definitions. Using the elements needed to describe a RMDP (Section 2.3), we define a *domain class* $\mathcal{D}_C$ as the tuple $\langle P_S, P_A, \mathcal{B} \rangle$, where P_S is the set of state predicates; P_A the set of action predicates and $\mathcal{B}$ the background knowledge. This means that problems that can use the same predicates to describe its states and actions with the same background knowledge or, in other words, problems that possess the same spaces of abstract states $\mathcal{S}_{ab}$ and abstract actions $\mathcal{A}_{ab}$, are in the same domain class. When the set of objects and a transition function are added, we obtain $\langle C, P_S, P_A, \mathcal{B}, \mathcal{S}, \mathcal{A}, T \rangle$, which specifies a *domain* $\mathcal{D}$. It describes the dynamics of the world and also the number of states and actions are now fixed. Finally, a *task* Ω is a tuple $\langle \mathcal{D}, R, \mathcal{G}, b^0 \rangle$, where $\mathcal{D}$ is a domain, R is the reward function, $\mathcal{G}$ is the set of goal states that indicates which states of the domain are desirable and b^0 indicates the initial state distribution. A task is fully described by an RMDP.

In this work we focus on knowledge transfer among different tasks within the same domain class. This means that tasks with states (and actions) described by the same predicates, but with different objects, are eligible. Research on using past knowledge to improve agent's performance in reinforcement learning is gaining prominence over the last years. This field of study is usually called *Transfer Learning* (TL), i.e., the agent learns the solution to some *source task* and uses this knowledge to somehow improve its performance when learning a *target task*.

The transfer learning problem we want to solve is described as follows.

Problem There is a set of source tasks $\Omega_1, \Omega_2, \dots, \Omega_n$, all from domain class $\mathcal{D}_C$, which the agent must solve using reinforcement learning (their transition function T_n and reward function R_n are unknown). Then, the agent faces a new target task Ω_{tg} , from the same domain class $\mathcal{D}_C$ and with unknown T_{tg} and R_{tg} as well. The objective of the agent is to solve Ω_{tg} with reinforcement learning using knowledge acquired from the source tasks, showing a better performance if compared to solving Ω_{tg} without any previous knowledge.

The goal of transfer learning is to improve learning in the target task by leveraging knowledge from source tasks. This improvement is usually observed by at least one of two characteristics, when comparing the performance of an agent with transfer with an agent without. The first is a *higher start*, or jumpstart, and indicates that an agent with transfer has an initial performance better than an agent without, before learning starts. The transfer agent starts with transferred knowledge while the agent without transfer is ignorant, hence the jumpstart. The second is a *higher slope*, which indicates a better performance in the beginning of the learning process. As the transferred knowledge is

expected to guide the agent to the most promising states, performance increases faster in the beginning. This can also be seen as a reduced time to achieve a certain threshold in performance. Given a threshold, agents with transfer tend to achieve it sooner than agents without. Figure 3.1 illustrates these measures.

Figure 3.1 – Three characteristics an agent with transfer learning might show when compared to an agent without transfer. Source: Torrey and Shavlik (2009)

There is also the possibility that a transfer learning method in fact decreases performance, instead of improving it, when compared to an agent without any transfer. This is called *negative transfer*. One of the challenges of the transfer learning methods is to provide positive transfer while avoiding negative transfer. In other words, positive transfer generally occurs when source and target tasks are appropriately related whereas negative transfer occurs when they are less related and it is up to the agent to detect these differences.

3.2 Related work

This section surveys past and current research related to the transfer learning problem presented in the previous section. The idea behind transfer learning (TL), to generalize solutions across tasks to improve learning, is not new and has been studied since a long time in the scope of human learning (THORNDIKE; WOODWORTH, 1901). However, TL for machine RL tasks has only recently been gaining attention from researchers of the AI community (TAYLOR; STONE, 2009).

When researching about TL, there are some important dimensions that the algorithms differ, each dimension highlighting different aspects of the problem. The main dimensions are:

1. *Allowed Task differences* – what are the allowed differences between source and target tasks? Many works focus only on transfer between tasks with the same states

and action spaces, whereas others allow them to be different. Allowing transfer to occur between less similar source and target tasks gives more flexibility to the agent.

2. *Source tasks* – do the TL algorithm allow just one source task or multiple? Some algorithms take into account just one source task, usually picked by a human, while others are concerned in gathering knowledge from a number of source tasks. If they allow multiple source tasks, they must also provide some mechanism to select the most relevant ones.
3. *Transferred knowledge* – what type of information is transferred between source and target task? This concerns on how knowledge is encoded for being transferred. Two most common types of transferred knowledge are value functions and policies. Other popular approach is to transfer partial policies, which may be used to solve parts of the target task.
4. *Learning methods* – what kind of algorithms are applicable to the agent? Agent might use TD methods, policy search, etc.

Allowed Task differences dimension classifies TL methods regarding the differences that can exist between the source task and target task. The simplest methods allow transfer between MDPs with different transition function (T), reward function (R) or initial state probability distribution (b^0). In these cases, the state and action spaces are the same between the source and target tasks. Other methods allow a broader transfer with different state and action spaces, given some restrictions. Methods that use a relational representation may allow the number of objects to change (C) or some just restrict that states and actions are described by same predicates (or features), allowing tasks to have different domains ($\mathcal{D}$) inside the same domain class. Different state and action spaces can also be achieved assuming there is a mapping function (map) between source and target spaces.

Dimension *Source tasks* concerns about the number of source tasks used and how they are selected. Most TL methods assume that a human has chosen a source task and this single task will be used for TL (h). Some other methods allow knowledge to come from multiple source tasks; they can just use them all for transfer (all) or just keep a subset of the most relevant tasks (lib).

TL methods have different *transferred knowledge* between source and target tasks. The transfer of learned value functions (V or Q) is popular, usually using them to initialize the function for the target task. Policies (π) are also transferred, and a quite prominent approach is the transfer of partial policies, or options (π_p), that are policies for subtasks (more detailed in Section 3.2.2). Transfer of policies can also be of abstract policies (π_{abs})

or stochastic abstract policies (π_{abs}^{stoch}). Additionally, we survey some methods that use case-based reasoning, thus transferring cases, that can be described as a tuple $\langle s, a, r \rangle$ (cases), where s is a state, a an action and r the expected reward of this case.

The type of transferred knowledge directly affects what are the *learning methods* allowed for the agent. If value functions are transferred, it is most likely that the agent uses a temporal-difference method (TD). If the agent transfers a policy, then a direct policy search (PS) can also be used. Other methods use some different kinds of learning methods such as: hierarchical learning (H), relational reinforcement learning (RRL) and case-based reasoning (CBR). Further details about each of these methods can be found in Sections 3.2.2, 3.2.3 and 3.2.5, respectively. Some TL methods focus on batch learning (Batch) or dynamic programming (DP) rather than online learning.

The following sections reviews some TL methods with different values for each of these dimensions and a summary of all methods described can be found in Table 3.1.

3.2.1 Transfer methods with same state and action spaces

We begin examining methods used for knowledge transfer between tasks with the same set of states and the same set of actions. Source and target tasks can thus differ on the transition function T , the reward function R or b^0 .

One of the earliest works in TL with RL demonstrates transfer learning in the 1-D pole balancing task changing the transition function T (SELFRIDGE; SUTTON; BARTO, 1985). The agent first learned how to balance a light and long pole (source task). Once the agent successfully learned it, the task was made harder (by changing T), with a heavier and shorter pole (target task). The total time spent learning on the sequence of two tasks and reusing the weights learned on the first task was actually shorter than just learning the hardest task.

Mehta et al. (2008) allow the agent to learn on a sequence of related tasks which are identical except for the reward function R . Their Variable-Reward Reinforcement Learning stores value functions for several MDPs and uses them to initialize the value function for a new task.

The idea of *learning from easy missions* (ASADA et al., 1996) also presents the agent with a set of tasks with increasing difficulty. This work applies Q-Learning on a shooting robot that needs to score a goal. The set of tasks needs to be built by an human, who moves the initial state s_0 (i.e. changes b^0) each time further from the goal. The agent learns how to reach the goal faster than if it had faced the full task directly. As only the initial state is changed and the state and actions spaces are the same, the Q function is defined over the same space and its values are also the same for all tasks.

3.2.2 Hierarchical learning

To bring more flexibility to transferring knowledge, applying some kind of abstraction over knowledge makes it easier to be generalized and thus used in transfer learning. Mehta et al. (2008), mentioned earlier, combines TL with *hierarchical reinforcement learning* (DIETTERICH, 2000), which involves breaking the MDP into a hierarchy of subproblems or subtasks. This hierarchy creates levels of abstraction. The abstraction can be a state abstraction, aggregating several states into one abstract state, or a temporal abstraction, in which sequences of actions are seen as a single decision. Then knowledge is transferred across tasks that share the same hierarchy. A potential drawback of hierarchical methods is that the learned policy may be suboptimal at the higher levels in the hierarchy.

When transfer learning topic is being discussed, one of the most cited works to present date is the use of *options* (SUTTON; PRECUP; SINGH, 1999). This work explores the use of *temporal abstraction* to improve performance on reinforcement learning. Options, also called macro-actions, partial policies or skills, are closed-loop policies – they are generalization of primitive actions in the form of temporally extended courses of actions. In other words, options are crafted to be similar to actions, adding the possibility they are temporally extended, and the agent can decide among options instead of actions (primitive actions can also be considered a single-step option). When the agent decides to execute an option, it unleashes a sequence of primitive actions, that guides the agent for more than just one time step. When using options instead of ground primitive actions, the decision problem is referred as a semi-Markov Decision Process (SMDP). Options are a framework for *temporal abstraction*, not an actual TL method. There are several methods which we review in this chapter that make use of options in transfer learning.

Example 5 Suppose a grid-like navigation task with 5 rooms (each with 9 cells) arranged in a row next to each other, with one door between each two adjacent rooms (total of 4 doors), as seen in Figure 3.2. The agent starts at the leftmost room (room 1) and has to reach the rightmost room (room 5). The agent’s primitive actions are to move one cell to north, south, east or west. An interesting set of options for this problem would be sequences of actions that take the agent from any point in a room to the door. There would be a total of 8 options, one for each pair room-door (one for taking the agent from room 1 to door connecting rooms 1-2; another option for taking the agent from room 2 to door connecting rooms 2-3; etc.). Learning to decide over the options set is faster because with a few iterations, the agent can be close to the goal. Conversely, if each iteration the agent takes just one primitive action (one step), much more iterations are necessary to reach the last room.

Figure 3.2 – Simple navigation environment with 5 rooms, numbered 1 to 5.

Asadi and Huber (2007), for example, make use of options and state abstraction in transfer learning. The proposed learning architecture has two levels: a decision-level and an evaluation-level. The agent automatically detects for subgoals (states that are visited more often) to learn options to reach these subgoals at the decision-level. The decision-level also has a compact state representation and uses Q-Learning. The evaluation level keeps the full state and action spaces and it is used to find discrepancies at the decision-level and increase its complexity if needed. The policy at the decision level is within a fixed bound of the optimal policy and this model is called a *Hierarchical Bounded Parameter SMDP*. After learning on the source task, the learned options and the decision-level representation are transferred to the target task.

Konidaris, Scheidwasser and Barto (2012) also proposes a TL method with state abstraction and use of options. It enables transfer via *shared features*, where features are functions related to the sensors the agent has. States are represented by a set of features, and transfer is made across tasks that are described by the same feature set. This enables transfer between tasks with different state spaces, unlike all works described so far. However, it is assumed that the action space remains the same, or at least that the source action space is a subset of the target action space. The proposed framework is applied with two kinds of transfer: reward shaping function transfer and skill transfer. *Reward shaping* (COLOMBETTI; DORIGO, 1993) refers to artificially creating another reward function to train the agent, accelerating RL. Shaping transfer is another paradigm for TL but this type of method is not particularly the focus of this work. In skill transfer, the agent can learn portable skills across a sequence of tasks that significantly improve performance on later related tasks. Experiments are made on several domains, including a robotic navigation domain.

3.2.3 Relational representation

Seeking to expand the idea of transferring knowledge to transfer across tasks with *different state (and action) spaces*, Dzeroski, De Raedt and Driessens (2001) propose to use a relational representation in RL tasks, giving rise to *Relational Reinforcement Learning* (RRL)

and an algorithm named Q-RRL. Due to the use of variables in relational representations, it is possible to abstract from specific details of the learning tasks, such as the specific goal pursued, and to exploit the results of previous learning phases when addressing new (more complex) situations. RRL addresses the *generalization* problem in RL and uses an MDP with relational representation, with definition similar to an RMDP (although this name is not used). Actually, the task is defined in a way that the state and action spaces are abstract and there is a fixed transition function for the abstract space as well.

Q-RRL algorithm is obtained by combining the classical Q-learning algorithm with stochastic selection of actions and a relational regression algorithm, TILDE-RT (BLOC-KEEL; DE RAEDT, 1998), which is an algorithm of induction of logical regression trees. Instead of having an explicit lookup table for the Q-function, an implicit representation of this function is learned in the form of a logical regression tree, called a Q-tree. Experiments with tasks on the blocks world were performed and RRL was effectively used to find abstract policies for stacking or unstacking with a varying number of blocks. Algorithm 7 shows the Q-RRL algorithm.

Algorithm 7 Q-RRL

```

▷ Given discount factor  $\gamma$ 
1: Initialize  $\hat{Q}_0$  to assign 0 to all  $(s, a)$  pairs
2: Initialize Examples to the empty set
3:  $e \leftarrow 0$ 
4: repeat
5:    $e \leftarrow e + 1$ 
6:    $i \leftarrow 0$ 
7:   while  $s_i \notin \mathcal{G}$  do
8:     Select an action following a strategy based on  $\hat{Q}_e$ 
9:     Perform action  $a_i$ 
10:    Receive immediate reward  $r_i$ 
11:    Observe new state  $s_{i+1}$ 
12:     $i \leftarrow i + 1$ 
13:    for  $j = i - 1$  to 0 do
14:       $\hat{q}_j = r_j + \gamma \max_{a'} \hat{Q}_e(s - j + 1, a')$ 
15:      Generate example  $x = (s_j, a_j, \hat{q}_j)$  and add  $x$  to Examples (replace if existent)
16:    Update  $\hat{Q}_e$  using TILDE to produce  $\hat{Q}_{e+1}$  using Examples
17: until some stop criterion is met
18: return  $\hat{Q}_e$ 

```

TILDE (for classification) and TILDE-RT (for regression) are inductive algorithms that make abstraction of the experiences, creating a first order logical decision tree. These

trees are an adaptation of decision trees for first order logic where the tests in the nodes are conjunctions of first order atoms. Apart from that, they are similar to classical decision trees algorithms such as C4.5 (QUINLAN, 1993). In fact, the heuristics, post-pruning algorithms, etc. employed by TILDE are exactly the same as C4.5. TILDE-RT minimizes the variance of the target variable within each subnode and maximizes the variance among two subnodes. The TILDE and TILDE-RT algorithms can be seen in Algorithm 8.

Algorithm 8 TILDE / TILDE-RT

```

1: procedure TILDE( $E$ : examples)
2:   Create a root node  $n$  for the tree  $t$ 
3:   split( $n, E, t$ )
4:   return  $t$ 
5:
6: procedure split( $n$ : node,  $E$ : examples,  $t$ : tree)
7:    $best \leftarrow \text{false}$ 
8:   for all possible tests  $q$  in node  $n$  do
9:     compute quality( $q$ )
10:    if quality( $q$ ) is better than quality( $best$ ) then
11:       $best \leftarrow q$ 
12:  if  $best$  yields improvements then
13:    test( $n$ )  $\leftarrow best$ 
14:    Create two subnodes  $n_1, n_2$  of  $n$  in  $t$ 
15:     $E_1 = \{e \in E | e \text{ satisfies } best \text{ in } t\}$ 
16:     $E_2 = \{e \in E | e \text{ does not satisfy } best \text{ in } t\}$ 
17:    split( $n_1, E_1, t$ )
18:    split( $n_2, E_2, t$ )
19:  else
20:    Turn  $n$  into a leaf
21:

```

Pursuing further development in theory of RRL, Kersting, van Otterlo and De Raedt (2004) introduce a relational update Bellman operator called REBEL. Bellman backup operator is a key concept in traditional RL (used in Q-Learning and Sarsa, for example). REBEL is used to create a relation value iteration algorithm (model-based) for RMDPs and it was effective in experiments with blocks world tasks. The authors also alert that value-based methods for RMDPs may not converge because an infinite number of abstract states has to be represented.

Croonenborghs, Driessens and Bruynooghe (2007) extends the use of the options framework to the relational setting to enable transfer learning across similar, but different, domains. Abstraction is achieved by using variables instead of constants in the relational representation and relational skills, represented by options, are learned to be transferred. Relational skills are interesting for the generalization they convey. These skills are learned using Q-RRL algorithm with TILDE as well.

Example 6 Consider the same task as Example 5. A relational option can generalize sequences of actions to go to the door. A single relational option `goToDoor(X)` can be used to express the ability to go to any door starting from anywhere inside any room.

Torrey (2009) also focus on transferring relational knowledge that guides action choices, motivated by the fact that relational knowledge typically can express information about relationships among objects and can use variables that generalize over classes of objects, making it more effective for transfer. Her work contributes with transfer algorithms in three categories: advice-based transfer, macro transfer and MLN transfer. Advice-based transfer uses source-task knowledge to provide advice for a target-task learner, which can follow, refine, or ignore the advice according to its value. *Advice-based transfer* is a different paradigm for transfer learning, which we do not focus in this work. Macro-transfer and MLN-transfer methods use source task experience to demonstrate good behavior for a target-task learner. The first transfers *macro-actions*, which are sequences of ground actions that were successful in the source task. The latter transfers either the Q function or a policy encoded in a Markov Logic Network (RICHARDSON; DOMINGOS, 2006). These transfer algorithms were evaluated experimentally in the reinforcement-learning domain of RoboCup simulated soccer and all of them provide empirical benefits compared to non-transfer approaches, either by increasing initial performance or by enabling faster learning in the target task.

Matos et al. (2011b) extended the idea of the TILDE algorithm to transfer an abstract policy encoded in the form of a first order logical decision tree for other tasks. The authors proposed algorithm ND-TILDE as an extension to TILDE to include more than one action at each leaf, thus building a stochastic abstract policy from examples in a source task (using the RMDP formalism as well) and then transferring it to a target task. This approach, however, is a *batch learning* method, i.e., it stores a number of interactions and then use the data for learning. It first learns an optimal policy in the ground source RMDP, with any algorithm. Once it is learned, set of experiences with the optimal policy is generated and ND-TILDE is applied to make an abstraction of these experiences, inducing a logical decision tree.

Qab-Learning (BEIRIGO et al., 2012) is an extension of the popular Q-Learning (Section 2.7.2). The basic idea is to apply the same update step from Q-Learning, but with abstract state and abstract action spaces. So, instead of learning the optimal Q function, it learns an “optimal” $Q_{ab} : \mathcal{S}_{ab} \times \mathcal{A}_{ab}$ function, which is the abstract Q function. Qab-Learning was tested in a robotic navigation domain and compared with ND-TILDE algorithm. Both algorithms have comparable results.

3.2.4 Inter-task mappings

Another approach to enable transfer across tasks with different state and actions spaces is to use *inter-task mappings*, functions that map the source state space to the target state space and the source action space to the target action space.

Taylor, Stone and Liu (2007) describes the TVITM (Transfer via Inter-task mappings) method, which makes transfer possible with these mappings. It actually uses three mappings: two from target states and actions to source states and actions and one from source Q function to target Q function. Once learned, the Q function for the source task is saved. Then, TVITM uses a TD method to compute Q for the target task, but its Q updates are a combination of the source task’s saved function with the target task’s current function. All mappings in this work are given, they are previously hand-coded by humans.

In addition to using temporal-difference methods, Taylor, Whiteson and Stone (2007) notes that it is important also to have TL methods with *policy search*. Policy search (PS) methods search directly in the space of all policies for the optimal one, without learning value function. There are some tasks that policy search methods can outperform TD methods (TAYLOR; WHITESON; STONE, 2006). The authors modify TVITM to a method capable of transferring policies, which are built through PS methods. Besides, apart from using hand-coded mappings, inter-task mappings are also learned from observations of both source and target tasks and applying a classifier over these observations.

Fernández, García and Veloso (2010) also presents a method for transferring (deterministic) policies with inter-task mappings, namely Policy Reuse. It uses Q-Learning with an exploration strategy that bias the learning of a new policy with one past policy. The probability of reusing the past policy starts high for the first step in an episode and decays exponentially. A policy can be reused on any other task given the mappings for state and action spaces.

Celiberto Jr. et al. (2011) combines *case-based reasoning* (CBR) with heuristics to achieve TL. CBR is a process in which inferences about a situation are drawn from individual instances called cases. The source task is learned with Q-Learning and it is built a case-base, where each case consist of a decision in a single step: a single action to be taken in a state. Then, the actions from the source task are mapped to actions on

the target task by a learning process with random simulations on both tasks (function mapping for states is hand-coded). Finally, learning on the target task uses CB-HAQL algorithm (BIANCHI; ROS; MANTARAS, 2009), which modifies the action selection on the exploitation of an ϵ -greedy strategy with an heuristic function calculated with the best case retrieved, for each time step.

3.2.5 Multiple source tasks

Most research on TL assumes a single source task has been learned and that this task is picked by a human, thereby assuring the agent should use it for transfer (and negative transfer is unlikely). However, some TL algorithms allow the agent to learn multiple source tasks and then use knowledge from all (or some) of them for transfer. Using knowledge from multiple tasks (not picked by humans) can be challenging especially because the solution of some source tasks might be relevant for the target task whereas others might not.

Fernández and Veloso (2006) introduce the idea of building a policy library, which contains policies for the source tasks, but a policy is included in the library only if it is sufficiently different from the ones already inside. The algorithm for building the library is the PRPL algorithm and it is further detailed in Section 4.3.1. Then, when learning a target task, the usefulness of each policy in the library is estimated by the average reward received after using each one and the most useful ones are transferred more often. Experiments with tasks with different goal states are made in a robotic navigation domain.

Mehta et al. (2008) uses a hierarchical RL method to learn subtask policies. When a novel target task is assigned to the agent, the agent sets the initial policy to that of the most similar source task, as a starting point for the policy. It uses a mechanism similar to PRPL to decide whether to include a new policy in the set of past policies or not (just includes if the gain of the it surpasses some threshold).

Martín and Geffner (2004) explored a different approach by creating *generalized policies*, which are policies that, based on a number of solved problem instances, are suitable to solve any problem in a domain. A generalized policy is a single set of rules that generalizes the solutions of a number of solved instances.

Generalized policies are also explored by Silva, Pereira and Costa (2012). The authors propose the AbsProb-PI algorithm, which deals with a set of source tasks in conjunction, as if all source tasks formed a single MDP. The algorithm then finds the best solution that maximizes the average return of all tasks, building a stochastic abstract policy. The proposed method uses policy search by performing a gradient-based policy iteration, but the algorithm is model-based and suitable only for offline learning.

Table 3.1 – Transfer learning methods discussed in Section 3.2, classifying each in terms of four dimensions: allowed task differences, source tasks, transferred knowledge and learning methods. The methods are in order of appearance in the text and key to abbreviations are in Table 3.2.

Citation	Allowed Task Differences	Source Tasks	Transferred Knowledge	Learning Methods
<i>Same state and action spaces</i>				
Selfridge, Sutton and Barto (1985)	T	h	Q	TD
Mehta et al. (2008)	R	lib	π_p	H
Asada et al. (1996)	b^0	h	Q	TD
<i>Hierarchical learning</i>				
Asadi and Huber (2007)	R	h	π_p	TD,H
Konidaris, Scheidwasser and Barto (2012)	$\mathcal{D}$	h	π_p	TD,H
<i>Relational representation</i>				
Dzeroski, De Raedt and Driessens (2001)	C	h	Q	RRL,TD
Kersting, van Otterlo and De Raedt (2004)	C	h	V	RRL,DP
Croonenborghs, Driessens and Bruynooghe (2007)	C	h	π_p	RRL,TD
Torrey (2009)	map	h	π_p, Q	TD
Matos et al. (2011b)	$\mathcal{D}$	h	π_{abs}^{stoch}	Batch
Beirigo et al. (2012)	$\mathcal{D}$	h	Q	TD
<i>Inter-task mappings</i>				
Taylor, Stone and Liu (2007)	map	h	Q	TD
Taylor, Whiteson and Stone (2007)	map	h	π	PS
Fernández, García and Veloso (2010)	map	lib	π	TD
Celiberto Jr. et al. (2011)	map	h	cases	CBR,TD
<i>Multiple source tasks</i>				
Fernández and Veloso (2006)	R, b^0	lib	π	TD
Martín and Geffner (2004)	C	all	π_{abs}	DP
Silva, Pereira and Costa (2012)	$\mathcal{D}$	all	π_{abs}^{stoch}	PS,DP
<i>Proposed methods in this work</i>				
S2L-RL with AbsSarsa(λ)	$\mathcal{D}$	lib	π_{abs}	TD
S2L-RL with AbsProb-RL	$\mathcal{D}$	all,lib	π_{abs}^{stoch}	PS

Table 3.2 – List of abbreviations used in Table 3.1.

Allowed Task differences	
b^0	Initial state probability distribution may differ
C	Objects (constants set) may differ
$\mathcal{D}$	Domain may differ within same domain class
R	Reward function or goal states may differ
T	Transition function may differ
map	Anything may differ, given a mapping function
Source tasks	
all	All source tasks are used
h	Only one human-selected source task is allowed
lib	Library with a subset of all source tasks
Transferred knowledge	
Q	Action-value function
V	State-value function
π	Deterministic ground policy
π_{abs}	Deterministic abstract policy
π_{abs}^{stoch}	stochastic abstract policy
π_p	Partial policy (option)
cases	Cases of type $\langle s, a, r \rangle$, where s is a state, a is an action and r is the expected reward
Learning methods	
Batch	Batch learning
CBR	Case-based reasoning
DP	Dynamic programming
H	Hierarchical learning
PS	Policy search
RRL	Relational reinforcement learning
TD	Temporal-difference

3.3 Discussion

As we have seen on the previous section, there is a number of existing TL methods, each with its own characteristics and restrictions. Table 3.1 shows all methods described here and we also included for comparison the two methods we propose in this work.

Some TL methods just allow transfer between tasks of the same domain. To grant more flexibility to the transfer, it is interesting to allow transfer between tasks with different state (and action) spaces. Mapping functions are a solution to grant flexibility, but they often require somebody to hand-code them for each pair of source and target tasks. Learning mapping functions is also possible, but it requires the agent to first interact with both tasks before. A relational representation, despite having its predicates hand-coded as well, partially overcomes this problem since this allows us to abstract over specific object identities and the number of objects involved, providing an automatic mapping among all tasks that are represented by the same relational alphabet.

In order to enable this knowledge transfer with different spaces, we explore the power of abstraction given by a relational representation. As seen on the previous section, abstraction is a good way to provide knowledge generalization. By representing knowledge from source tasks in a higher level of abstraction, this knowledge can be applied to another task that is similar. In this work, abstraction is given by design, being directly related to the predicates that describe states and actions of a RMDP and following a fixed rule: it is achieved by replacing constants for variables in the predicates.

The problem we want to solve that we previously defined states that knowledge comes from a set of source tasks. However, the majority of TL works deals only with a single source task. Both methods we propose use the idea of using a policy library and S2L-RL with AbsProb-RL also present a method for finding a generalized policy for all source tasks. Some methods that we presented tackle a sequence of tasks, which is also interesting. These sequences can be useful in *Lifelong learning* (THRUN; MITCHELL, 1995), a TL-related paradigm in which the agent experience a sequence of tasks and keeps building knowledge continuously. Actually, the agent might not be told when a task begins. TL is a key component to lifelong learning systems.

Regarding the transferred knowledge, value functions and policies are the most common choices. Policies are more general than value functions, since we derive policies from value functions but cannot do the opposite. Thus, they generalize better to other tasks. We seek to transfer knowledge in quite general cases (same domain class), then policy transfer is more suitable. The level of abstraction shared in a domain class can be too high to provide a good abstraction for value functions. That is why we opt for transferring policies; more specifically, abstract policies.

For the learning methods, we use online learning algorithms (unlike batch algorithms such as TILDE) for building the abstract policies. In addition, AbsProb-RL uses direct policy search, which might bring benefits to the resulting policy (more details in Section 4.2.3) and it is not usually found in TL works.

To sum up, the agent is going to use reinforcement learning to find a policy π_{tg} to accomplish this new task Ω_{tg} , but we want it to use past knowledge (abstract policies) in order to improve performance. Given this problem, there are two questions that arise:

1. How can we build abstract policies from the set of source tasks?
2. How can the agent use these policies to improve its performance in the target task?

Following chapters presents our contributions to answer these questions. Chapter 4 shows two algorithms to build abstract policies (AbsSarsa(λ) and AbsProb-RL) and chapter 5 introduces an architecture (S2L-RL) that enables the agent to incorporate past policies in the learning process of a task.

4 BUILDING ABSTRACT POLICIES

This chapter presents methods to build abstract policies with reinforcement learning in an RMDP. The main purpose of building abstract policies is to generalize the solution of a problem so that the abstract policies can be used for improving the agent’s performance in an upcoming target task. Here we propose two methods: Section 4.1 presents AbsSarsa(λ), based on learning a value function; and Section 4.2 presents AbsProb-RL, a policy-search algorithm.

4.1 Building abstract policies from action-value function

Our first algorithm for building an abstract policy is an extension of the popular Sarsa(λ) algorithm (Section 2.7.3). The inspiration came from a published work (BEIRIGO et al., 2012) (which I co-authored) that presented the *Qab-Learning* algorithm (refer to Section 3.2.3). The issue with Qab-Learning is that there are no guarantees that working with Q function at the abstract level converges to a proper policy, assuming that there is at least one (a proper policy is a policy that reaches the goal *w.p.1* from all states, with time tending to infinity). The MDP is defined at the ground level and the Markov property may not hold when lifting it to the abstract level. Despite this limitation, there are certain cases that a very good policy, if not the best, can be found.

This limitation of not holding the Markov property given by the abstract level is similar to the problem faced in POMDPs. The agent’s partial observations prevents the agent from observing the exact states. Possible approaches to overcome this limitation include doing state estimation and creating belief states or simply ignoring the hidden states and working directly at the observation level (ABERDEEN, 2003). These observations are similar to abstract states because both of them carry less information than a ground state, thus representing an abstraction to the state space. And thereby, the Markov property may not exist at their level. We can apply methods for solving POMDPs (LOCH; SINGH, 1998; LI; YIN; XI, 2011) in solving RMDPs at the abstract level.

Loch and Singh (1998) shows empirically that Sarsa(λ), an RL algorithm that uses eligibility traces, can work very well on POMDPs that have good memoryless policies, i.e., on problems in which although there is very poor observability, there also exists a mapping from the agent’s immediate observations to actions that yields near-optimal return. Inspired by this fact and Qab-Learning, we created AbsSarsa(λ), an extension to the Sarsa(λ) algorithm to work on the abstract level.

4.1.1 AbsSarsa(λ) algorithm

We propose to use the Sarsa(λ) algorithm, given in Algorithm 6, to learn an abstract policy. The objective is to derive a policy from the estimate of function $Q_{ab} : \mathcal{S}_{ab} \times \mathcal{A}_{ab}$. However, clearly only ground states are visited by the real system, and only ground actions can be actually applied. Learning must proceed by processing, at time t , the experience $\langle s_t, a_t, r_t, s_{t+1}, a_{t+1} \rangle$. This ground experience is related to the tuple $\langle \sigma_t, \alpha_t, r_t, \sigma_{t+1}, \alpha_{t+1} \rangle$ that needs to be used to update the eligibility trace function $\eta_{ab}^t(\sigma, \alpha)$ and the Q_{ab} -value estimate $Q_{ab}^t(\sigma, \alpha)$.

Now one can consider two distinct strategies. First, the agent can observe the ground states s_t and s_{t+1} and translate them to abstract states σ_t and σ_{t+1} by applying the abstraction operator $\phi_s : \mathcal{S} \rightarrow \mathcal{S}_{ab}$. Then it can choose abstract actions α_t and α_{t+1} based on the abstract states and current estimate of Q_{ab} . These abstract actions are then grounded to one of the possible ground actions to be executed. The agent receives reward, observes next state and then the cycle begins again (note that reward r_t does not undergo any transformation). This is what we call *AbsSarsa(λ) Active* and the resulting reinforcement learning scheme is given by Algorithm 9.

The second strategy assumes the existence of a “ground level” reinforcement learning scheme that is learning a (ground) policy, say using Sarsa(λ). Now there is a second learner at an “abstract level” which can observe the actions a_t and a_{t+1} taken by the ground level learner, and translate then into α_t and α_{t+1} (together with σ_t and σ_{t+1}) so as to run Sarsa(λ). The abstract level learner does not influence on the policy being executed; it merely observes a ground learner. This is what we call *AbsSarsa(λ) Passive*, or simply *AbsSarsa(λ)*, and the resulting reinforcement learning scheme is given by Algorithm 10.

To summarize, active abstraction runs reinforcement learning and directly specifies ground actions by translating abstract actions, while passive abstraction simply observes the experiences of interactions experienced by a ground level learner. Passive abstraction does not actually apply the abstract policy being learned in the task, but only receives the experiences from the interactions conducted at the ground level of learning and uses these experiences to update the Q_{ab} -values.

We now argue that passive abstraction is better than active abstraction for such purpose, since the former is an abstraction of the learning process conducted at the ground level, while the latter seeks to directly extract the structure of the task as built by a ground learner, which is much more complex given the difficulties derived from aggregations of state information made in the abstract level.

An experimental comparison of the two strategies for learning abstract policies is shown in Figure 4.1, and in this figure we can see that the results support our arguments. These experiments were performed in a grid-like robotic navigation domain class. In this

Algorithm 9 AbsSarsa(λ) Active

$\triangleright$ Given discount factor γ , decay rate λ and learning rate μ and maximum number of steps per episode t_{max}
 1: Initialize $Q_{ab}(\sigma, \alpha)$ arbitrarily for all $\sigma \in \mathcal{S}_{ab}$ and $\alpha \in \mathcal{A}_{ab}$
 2: **for** each episode **do**
 3:   (Re)initialize $\eta_{ab}(\sigma, \alpha)$ with 0 for all $\sigma \in \mathcal{S}_{ab}$ and $\alpha \in \mathcal{A}_{ab}$
 4:   Observe state s
 5:    $\sigma \leftarrow \phi_s(s)$ $\triangleright$ Find corresponding abstract state
 6:   Choose action α using a policy derived from Q_{ab}
 7:    $a = \text{grounding}(\alpha, s)$
 8:   **for** each episode step $t \in \{0, 1, \dots, t_{max}\}$ or until $s \in \mathcal{G}$ **do**
 9:     Take action a , observe new state s' and receive reward r
 10:      $\sigma' \leftarrow \phi_{s'}(s')$
 11:     Choose next action α' using a policy derived from Q_{ab} (e.g. ϵ -greedy)
 12:      $a' = \text{grounding}(\alpha', s')$
 13:      $\delta_{Q_{ab}} = r + \gamma Q_{ab}(\sigma', \alpha') - Q_{ab}(\sigma, \alpha)$
 14:      $\eta_{ab}(\sigma, \alpha) \leftarrow \eta_{ab}(\sigma, \alpha) + 1$
 15:     **for** all σ_i in $\mathcal{S}_{ab}$ and α_j in $\mathcal{A}_{ab}$ **do**
 16:        $Q_{ab}(\sigma_i, \alpha_j) \leftarrow Q_{ab}(\sigma_i, \alpha_j) + \mu \eta_{ab}(\sigma_i, \alpha_i) \delta_{Q_{ab}}$
 17:        $\eta_{ab}(\sigma_i, \alpha_j) \leftarrow \gamma \lambda \eta_{ab}(\sigma_i, \alpha_j)$
 18:      $\sigma \leftarrow \sigma'$
 19:      $\alpha \leftarrow \alpha'$
 20: **return** $\pi_{ab}(\sigma) \leftarrow \max_{\alpha \in \mathcal{A}_{ab}} Q_{ab}(\sigma, \alpha), \forall \sigma \in \mathcal{S}_{ab}$

---

experiment, the agent starts in a random initial state and has to complete the task to reach a goal location (5 different goal locations are used) in domain $\mathcal{D}_1$ 6.1. Its actions are only one-step-movements and it receives reward only when it reaches the goal. The agent runs 1000 episodes for each task, with a maximum of 500 steps in each episode. Further details about the robotic navigation domain class can be found in Section 6.1.

We draw three curves: *Ground level*, *AbsSarsa(λ) active* and *AbsSarsa(λ) (passive)*. The curve *Ground level* is the result of learning at the ground level using the standard Sarsa(λ) algorithm and an ϵ -greedy strategy, for reference as the optimal policy. The curves *AbsSarsa(λ) active* and *AbsSarsa(λ) (passive)* show the result of applying Algorithms 9 and 10, respectively. Each point in the curve represents the actual value W^π of the policy π learned up to that episode using each algorithm (Eq. 2.7). Ground level shows the value of a ground policy, whereas both AbsSarsa(λ) curves shows the value of an abstract policy.

Algorithm 10 AbsSarsa(λ) (Passive)

$\triangleright$ Given discount factor γ , decay rate λ and learning rate μ and maximum number of steps per episode t_{max}
 1: Initialize $Q_{ab}(\sigma, \alpha)$ arbitrarily for all $\sigma \in \mathcal{S}_{ab}$ and $\alpha \in \mathcal{A}_{ab}$
 2: **for** each episode **do**
 3:     (Re)initialize $\eta_{ab}(\sigma, \alpha)$ with 0 for all $\sigma \in \mathcal{S}_{ab}$ and $\alpha \in \mathcal{A}_{ab}$
 4:     **for** each episode step $t \in \{0, 1, \dots, t_{max}\}$ or until $s \in \mathcal{G}$ **do**
 5:         Observe $\langle s, a, r, s', a' \rangle$ from ground level Sarsa(λ)
 6:          $\sigma \leftarrow \phi_s(s)$; $\alpha \leftarrow \phi_a(a)$
 7:          $\sigma' \leftarrow \phi_s(s')$; $\alpha' \leftarrow \phi_a(a')$
 8:          $\delta_{Q_{ab}} = r + \gamma Q_{ab}(\sigma', \alpha') - Q_{ab}(\sigma, \alpha)$
 9:          $\eta_{ab}(\sigma, \alpha) \leftarrow \eta_{ab}(\sigma, \alpha) + 1$
 10:        **for** all σ_i in $\mathcal{S}_{ab}$ and α_j in $\mathcal{A}_{ab}$ **do**
 11:             $Q_{ab}(\sigma_i, \alpha_j) \leftarrow Q_{ab}(\sigma_i, \alpha_j) + \mu \eta_{ab}(\sigma_i, \alpha_i) \delta_{Q_{ab}}$
 12:             $\eta_{ab}(\sigma_i, \alpha_j) \leftarrow \gamma \lambda \eta_{ab}(\sigma_i, \alpha_j)$
 13: **return** $\pi_{ab}(\sigma) \leftarrow \max_{\alpha \in \mathcal{A}_{ab}} Q_{ab}(\sigma, \alpha), \forall \sigma \in \mathcal{S}_{ab}$

All curves used an ϵ -greedy strategy with a fixed value of $\epsilon = 0.1$, and also $\gamma = 0.95$, $\lambda = 0.9$, $\mu = 0.1$.

Note that active abstraction is significantly worse than passive abstraction. Indeed, AbsSarsa(λ) active, despite being the most straightforward adaptation of Sarsa(λ) to the abstract level, fails to effectively find an abstract policy. This is because when the abstract policy itself guides the whole learning, its behavior at the ground level may be erratic and unstable due to the lack of information lost in the abstraction. Conversely, the passive abstraction just applies abstraction over a policy that is guaranteed to converge. When it observes the ground level, it is observing a policy that increases its value monotonically, as Sarsa(λ) always improves its estimate of Q function.

The behavioral duality we notice between the ground and (passive) abstract level is *exactly* what we seek when building abstract policies for transfer learning. The abstract policy does not reach a value as high as the optimal ground policy, as it deals with a poorer representation. However, this smaller space allows the abstract learner to have a faster convergence, reaching maximum value sooner. In the initial episodes, the abstract policy actually achieves a higher value than the ground policy and this is the improvement this kind of knowledge can make to a reinforcement learning agent: better performance in the initial episodes. And one challenge is to combine both behaviors: good performance in the beginning with optimal performance in the end. This is discussed on Chapter 5.

Figure 4.1 – Comparison of policy values between learning in the ground level with the plain Sarsa(λ) algorithm (value of ground policy shown), and the approaches to active and passive abstraction with AbsSarsa(λ) (value of abstract policies shown).

4.1.2 Shortcomings of AbsSarsa(λ)

Recall that in Section 2.4 we presented a hierarchy of five state abstraction schemes based on which features they preserve: ϕ_s^{model} , $\phi_s^{Q^\pi}$, $\phi_s^{Q^*}$, $\phi_s^{a^*}$, and $\phi_s^{\pi^*}$. Summing up, ϕ_s^{model} preserves the one-step model; $\phi_s^{Q^\pi}$ preserves the state-action value function for all policies; $\phi_s^{Q^*}$ preserves the optimal state-action value function; $\phi_s^{a^*}$ preserves the optimal action and its value; and $\phi_s^{\pi^*}$ preserves the optimal action.

Li, Walsh and Littman (2006), besides defining the five abstractions, proved also some theorems regarding them, one of which we reproduce here:

Theorem Assume that each state-action pair is visited infinitely often and the step-size parameters decay appropriately.

1. Q-learning with abstractions ϕ_s^{model} , $\phi_s^{Q^\pi}$, or $\phi_s^{Q^*}$ converges to the optimal state-action value function in the ground MDP. Therefore, the resulting optimal abstract policy is also optimal in the ground MDP.
2. Q-learning with abstraction $\phi_s^{a^*}$ does not necessarily converge. However, if the behavior policy is fixed, Q-learning converges to Q^* with respect to some weighting $w(s)$, and the greedy policy is optimal in the ground MDP, although

Q^* may not predict the optimal values for suboptimal actions in the ground MDP.

3. Q-learning with abstraction $\phi_s^{\pi^*}$ can converge to an action-value function whose greedy policy is suboptimal in the ground MDP. However, we note that policy-search methods may still find the optimal policy in this case. (LI; WALSH; LITTMAN, 2006)

This theorem says that learning the Q function leads to the optimal policy with abstractions ϕ_s^{model} , $\phi_s^{Q^\pi}$, $\phi_s^{Q^*}$ and $\phi_s^{a^*}$. However, it is suboptimal for $\phi_s^{\pi^*}$, the coarsest abstraction listed. AbsSarsa(λ) is an algorithm that learns the Q function. In order to cover this more general case, it is pointed that a policy-search method may be more effective.

Ng (2003) also points out there are many MDPs with Q functions that are complicated and difficult to approximate and policy-search methods can more readily exploit this fact, given they work with policies directly, without the intermediate step of representing a value function. In the next section we propose a policy-search algorithm for building abstract policies.

4.2 Building abstract policies using policy search

Given the shortcomings of building an abstract policy based on a value function shown in the previous section, we would like to build an algorithm that searches directly through the set of all policies for the best one.

AbsProb-PI (SILVA; PEREIRA; COSTA, 2012) is a policy-search algorithm that can build stochastic abstract policies. It uses a gradient function to search in the policy space for the best stochastic memoryless policy for a task. However, it is a planning algorithm, which requires a complete model of the problem. This is not the case of the type of problems we want to tackle, which are reinforcement learning problems. We are thus interested in adapt AbsProb-PI algorithm to a RL problem, in which we note one main difference: part of the model is unknown (the transition and reward functions).

Before going into the algorithm, we discuss further about stochastic policies, because we have only dealt with deterministic policies so far. As AbsProb-PI builds stochastic abstract policies, it is important to understand the differences to deterministic ones and reasons why they were chosen. Then AbsProb-PI is summarized to provide the basis for a full understanding of its adaptation to RL, the AbsProb-RL algorithm.

4.2.1 Stochastic abstract policies

Recall that a *stochastic abstract policy* is defined as

$$\pi_{ab} : \mathcal{S}_{ab} \times \mathcal{A}_{ab} \rightarrow [0, 1], \text{ with } \pi_{ab}(\sigma, \alpha) = P(\alpha|\sigma), \sigma \in \mathcal{S}_{ab}, \alpha \in \mathcal{A}_{ab}, \quad (4.1)$$

which means that, for each state-action pair, there is a probability associated. Given a state σ , the sum of probabilities for all actions $\alpha \in \mathcal{A}_{ab}$ must be equal to 1. The higher the probability $\pi_{ab}(\sigma, \alpha)$, the more likely that action α is to be executed in state σ . Deterministic policies can be seen as a special case of stochastic policies, case that for each state, just one action has probability 1 and all others have probability 0.

Since abstract spaces aggregate states and actions, the Markov property at the ground level of an RMDP may not hold in an abstract level of the same RMDP (SILVA; PEREIRA; COSTA, 2012), being similar to POMDPs. In this case, stochastic policies are appropriate, because can be better than deterministic policies (SINGH; JAAKKOLA; JORDAN, 1994), as they are more flexible for offering more than one choice of action per state.

To illustrate the statement that stochastic policies are more appropriate for the abstract level, let us analyze a simple example. Consider a discrete environment with 6 cells in line, as Figure 4.2 shows, with goal on cell 4. Each cell represents a state and the agent can execute the following actions: go right, go left or stay still. The optimal deterministic policy for this problem would be:

- If the agent is in cells 1, 2 or 3: go right;
- If the agent is in cell 4: stay still;
- If the agent is in cell 5 or 6: go left.

Figure 4.2 – Simple discrete environment. Each cell represents a state; cell 4 is the goal and arrows indicate optimal policy.

Consider the agent has only a partial observability of the environment, being able only to perceive what is immediately on its surroundings. That is, it has the following perceptions:

- Perception on cell 1: [wall up, wall left, door right, wall down, not in goal];
- Perception on cell 2: [wall up, door left, door right, wall down, not in goal];

- Perception on cell 3: [wall up, door left, door right, wall down, not in goal];
- Perception on cell 4: [wall up, door left, door right, wall down, in goal];
- Perception on cell 5: [wall up, door left, door right, wall down, not in goal];
- Perception on cell 6: [wall up, door left, wall right, wall down, not in goal].

With these perceptions, the agent cannot distinguish states 2, 3 and 5. In fact, the agent can only perceive four distinct situations:

- Situation 1 = σ_1 = [wall up, wall left, door right, wall down, not in goal];
- Situation 2 = σ_2 = [wall up, door left, door right, wall down, not in goal];
- Situation 3 = σ_3 = [wall up, door left, door right, wall down, in goal];
- Situation 4 = σ_4 = [wall up, door left, wall right, wall down, not in goal].

These situations could also be four abstract states, given the correct abstraction. Given these four situation, which deterministic policy would solve (even if suboptimally) the problem? If the agent is in situation 2, which in fact means that the agent could be in cell 2, 3 or 5, neither going right nor going left solves the problem. In this case, a stochastic policy can solve it, even if suboptimally. A possible stochastic abstract policy for this problem would be:

- If in σ_1 , go right *w.p.* 1;
- If in σ_2 , go right *w.p.* 0.66 and go left *w.p.* 0.33;
- If in σ_3 , stay still *w.p.* 1
- If in σ_4 , go left *w.p.* 1.

Thus, this stochastic abstract policy presents a mean value of each state (considering all states equally probable) higher than any deterministic policy.

4.2.2 AbsProb-PI algorithm

Here we summarize AbsProb-PI (SILVA; PEREIRA; COSTA, 2012), a model-based algorithm that, given an RMDP, uses cumulative discounted reward evaluation to build an abstract policy.

AbsProb-PI is based on the popular Policy Iteration algorithm (Section 2.6), but it is designed to perform in an abstract level. At each iteration, the idea is to find an improved version of the current policy. A gradient function G is used to determine the

improvement direction, as in a gradient-based policy iteration (CAO; FANG, 2002). The policy improvement goes along the direction with the steepest policy value gradient.

As the abstract state space $\mathcal{S}_{ab}$ does not necessarily hold the Markov property, it is important to consider the initial state distribution b^0 . We must define the transition matrix $\mathbf{T}^{\pi_{ab}}$ of an abstract policy π_{ab} . Consider a matrix-vector representation of functions $b^0(s)$, $R(s)$, $V^\pi(s)$ and $T^\pi(s, s') = \sum_{a \in \mathcal{A}} \pi(a|s)T(s, a, s')$: $\mathbf{b}^0$, $\mathbf{R}$, $\mathbf{V}^\pi$ and $\mathbf{T}^\pi$, respectively. Being $\phi_s : \mathcal{S} \rightarrow \mathcal{S}_{ab}$ the function that maps ground states into abstract states and $\mathbf{I}$ the identity matrix, we define $\mathbf{T}^{\pi_{ab}}$ and $\mathbf{V}^{\pi_{ab}}$ by:

$$T^{\pi_{ab}}(s, s') = \sum_{\alpha \in \mathcal{A}_{ab}} \pi_{ab}(\phi_s(s), \alpha) \sum_{a \in \mathcal{A}_\alpha} \frac{1}{|\mathcal{A}_\alpha|} T(s, a, s'), \quad (4.2)$$

$$\mathbf{V}^{\pi_{ab}} = (\mathbf{I} - \gamma \mathbf{T}^{\pi_{ab}})^{-1} \mathbf{R}. \quad (4.3)$$

Note that we assume that all ground actions abstracted by an abstract action are chosen with uniform probability. In the AbsProb-PI algorithm we must define the parameter $\epsilon > 0$ which guarantees that the policy converges at most to an ϵ -greedy policy; we must also choose the step size used in the gradient descent method represented by the function $\delta_\pi(i)$. In our experiments we use a decreasing step size, i.e., $\delta_\pi(i) = \frac{1}{1+i}$, where i is the iteration step.

Algorithm 11 presents the AbsProb-PI algorithm. An abstract policy π_{ab} is initialized arbitrarily. Then the algorithm iteratively refines this policy, until some stopping criterion is met. The first step in each iteration (line 3) corresponds to the evaluation of the current policy. The product $\mathbf{C}$ (line 4) denotes the expected cumulative occurrence of each state, considering policy π_{ab} and b^0 . Next, for each abstract action α a policy $\pi_{\alpha, \epsilon}$ is defined. This policy chooses action α with probability $1 - \epsilon$ and chooses uniformly all other actions with probability ϵ . $\mathbf{T}^{\pi_{\alpha, \epsilon}}$ is the matrix-vector representation of the transition function of policy $\pi_{\alpha, \epsilon}$, defined similarly to Equation 4.2. The difference $\Delta^{\alpha, \pi_{ab}}$ each policy $\pi_{\alpha, \epsilon}$ causes in the value of each state is calculated (line 5). Then, G is calculated for each pair abstract state-abstract action (σ, α) by adding the values of $\Delta^{\alpha, \pi_{ab}}(s)$ weighted by $C(s)$, for all $s \in \mathcal{S}_\sigma$ (line 6). The value of G represents the gradient of the value function, and its maximum value indicates which abstract action should be executed more often to achieve a better result (line 7). The policy is then updated in this direction, using a step size δ_π (line 8), which can be changed in each iteration.

4.2.3 AbsProb-RL algorithm

We now want to adapt AbsProb-PI to build abstract policies in an *online* approach, i.e., for reinforcement learning problems. We must adapt the elements that need the transition

Algorithm 11 AbsProb-PI

-
- ▷ Given a task Ω , step size function δ_π , discount factor γ and exploration rate ϵ
 1: Initialize the stochastic abstract policy π_{ab} arbitrarily such that $\pi_{ab}(\sigma, \alpha) \geq \frac{\epsilon}{|\mathcal{A}_{ab}|} \forall \sigma \in \mathcal{S}_{ab}, \alpha \in \mathcal{A}_{ab}$ and $\sum_{\alpha \in \mathcal{A}_{ab}} \pi_{ab}(\sigma, \alpha) = 1 \forall \sigma \in \mathcal{S}_{ab}$
 2: **for** each iteration i **do**
 3:   Calculate the value function $\mathbf{V}^{\pi_{ab}}$
 4:   Calculate the product $\mathbf{C} = \gamma \mathbf{b}^{0\top} (\mathbf{I} - \gamma \mathbf{T}^{\pi_{ab}})^{-1}$
 5:   For each action $\alpha \in \mathcal{A}_{ab}$ calculate $\Delta^{\alpha, \pi_{ab}} = (\mathbf{T}^{\pi_{\alpha, \epsilon}} - \mathbf{T}^{\pi_{ab}}) \mathbf{V}^{\pi_{ab}}$
 6:   For each $\sigma \in \mathcal{S}_{ab}$ and $\alpha \in \mathcal{A}_{ab}$, calculate $G(\sigma, \alpha) = \sum_{s \in \mathcal{S}_\sigma} C(s) \Delta^{\alpha, \pi_{ab}}(s)$
 7:   For each $\sigma \in \mathcal{S}_{ab}$ find the best direction $\alpha_\sigma^* = \arg \max_{\alpha \in \mathcal{A}_{ab}} G(\sigma, \alpha)$
 8:   Update policy π_{ab} :
-

$$\pi_{ab}(\sigma, \alpha) \leftarrow \begin{cases} (1 - \delta_\pi(i))\pi_{ab}(\sigma, \alpha) + \delta_\pi(i) \left(\frac{\epsilon}{|\mathcal{A}_{ab}|} + (1 - \epsilon) \right) & , \text{ if } \alpha = \alpha_\sigma^* \\ (1 - \delta_\pi(i))\pi_{ab}(\sigma, \alpha) + \delta_\pi(i) \frac{\epsilon}{|\mathcal{A}_{ab}|} & , \text{ if } \alpha \neq \alpha_\sigma^* \end{cases}$$

---

function T to be calculated, as this function is not available to an RL learner. To do this we use Monte Carlo estimation with RL, combining AbsProb-PI with ideas from other works (JAAKKOLA; SINGH; JORDAN, 1994; WILLIAMS; SINGH, 1998) to present a new algorithm called AbsProb-RL. This is one of the main contributions of this work and is also a result of published work (KOGA; SILVA; COSTA, 2013a).

Looking at the AbsProb-PI (algorithm 11), we notice that the transition function is used on lines 3, 4 and 5, where AbsProb-PI calculates: $V^{\pi_{ab}}$, the value function of the current policy π_{ab} ; $\mathbf{C}$, the expected accumulate discounted occupation of each state $s \in \mathcal{S}$; and $\Delta^{\alpha, \pi_{ab}}$, the local gain of action α against policy π_{ab} when applied in each state $s \in \mathcal{S}$ and then following policy π_{ab} . Then at line 6 the ground state observation is used to combine $\mathbf{C}$ and $\Delta^{\alpha, \pi_{ab}}$ for each state $s \in \mathcal{S}$; and then the gradients of the ground states are aggregated with their corresponding abstract states. We show below that the gradient $G(\sigma, \alpha)$ can be calculated directly by a Monte Carlo approach, not needing thereby the transition function.

The use of the Monte-Carlo-based estimation implies that the agent must execute several interactions with the environment and the resulting set of experiences from these interactions allows it to estimate the gradient. This means that, to be equivalent to each iteration of AbsProb-PI, agent must execute a number of episodes with the current policy, a number of which is big enough to have a good estimate.

The gradient $G^{\pi_{ab}}(\sigma, \alpha)$ can be written as:

$$G^{\pi_{ab}}(\sigma, \alpha) = \sum_{s \in \mathcal{S}_\sigma} C(s) \sum_{s' \in \mathcal{S}} \left(T(s, \alpha, s') - \sum_{\alpha' \in \mathcal{A}_{ab}} \pi_{ab}(\sigma, \alpha') T(s, \alpha', s') \right) V^{\pi_{ab}}(s') \quad (4.4)$$

$$\begin{aligned} &= \sum_{s \in \mathcal{S}_\sigma} C(s) \sum_{s' \in \mathcal{S}} T(s, \alpha, s') V^{\pi_{ab}}(s') - \\ &\quad - \sum_{s \in \mathcal{S}_\sigma} C(s) \sum_{\alpha' \in \mathcal{A}_{ab}} \pi_{ab}(\sigma, \alpha') T(s, \alpha', s') V^{\pi_{ab}}(s') \end{aligned} \quad (4.5)$$

$$= G_{\text{alternative}}^{\pi_{ab}}(\sigma, \alpha) - G_{\text{current}}^{\pi_{ab}}(\sigma, \alpha), \quad (4.6)$$

and can be divided in two terms where each term is weight by $C(s)$. $C(s)$ is the expected accumulate discounted occupation of each state $s \in \mathcal{S}$ following π_{ab} and we have ($\mathbf{C}$ is the vector representation of $C(s)$):

$$\mathbf{C} = \gamma \mathbf{b}^{0\top} (\mathbf{I} - \gamma \mathbf{T}^{\pi_{ab}})^{-1} \quad (4.7)$$

$$= \gamma \mathbf{b}_{0\top} (\mathbf{I} + \gamma \mathbf{T}^{\pi_{ab}} + \gamma^2 (\mathbf{T}^{\pi_{ab}})^2 + \dots) \quad (4.8)$$

$$= \gamma \mathbf{b}_{0\top} \sum_{t=0}^{\infty} \gamma^t (\mathbf{T}^{\pi_{ab}})^t. \quad (4.9)$$

Since $\mathbf{b}_0(\mathbf{T}^{\pi_{ab}})^t(s) = P(s_t = s | \pi_{ab})$, we define:

$$C(s) = \gamma \sum_{t=0}^{\infty} \gamma^t P(s_t = s | \pi_{ab}) \quad (4.10)$$

and $C(s)$ can be estimated by executing π_{ab} and counting state occurrence, i.e.,

$$\hat{C}(s) = \gamma \frac{\sum_{h=1}^H \sum_{t=0}^{t_h-1} \gamma^t \mathbb{1}_{\{s\}}(s_{t,h})}{|H|} \quad (4.11)$$

where H is the number of episodes considered for the estimation, t_h is the size of the episode h , $s_{t,h}$ is the state visited at time t in the episode h , and $\mathbb{1}_{\{s\}}(x)$ is the indicator function:

$$\mathbb{1}_{\{s\}}(x) = \begin{cases} 1 & \text{if } x \in \{s\}, \\ 0 & \text{otherwise.} \end{cases} \quad (4.12)$$

As mentioned before, $C(s)$ weights expected value of two factors: rewards obtained from each state by executing current abstract policy, and rewards obtained from each state after executing an arbitrary action and following current abstract policy.

In the first case, since the executed policy is the same that must be sampled, all that must be done is to sample the obtained reward after occurrence of each state, i.e.,

$$\hat{G}_{current}^{\pi_{ab}}(\sigma, \alpha) = \gamma \sum_{s \in \mathcal{S}_\sigma} \frac{\sum_{h=1}^H \sum_{t=0}^{t_h-1} \gamma^t r_{t,h} \sum_{\bar{t}=0}^t \mathbb{1}_{\{s\}}(s_{\bar{t},h})}{|H|} \quad (4.13)$$

$$= \gamma \frac{\sum_{h=1}^H \sum_{t=0}^{t_h-1} \gamma^t r_{t,h} \sum_{\bar{t}=0}^t \mathbb{1}_{\mathcal{S}_\sigma}(s_{\bar{t},h})}{|H|}, \quad (4.14)$$

where $r_{t,h}$ is the reward received at time t in episode h .

In the second case, an arbitrary abstract action α must be experimented with probability one, i.e., after the occurrence of the state s , the next state s' must be drawn from $P(s'|s, a) = T(s, a, s')$ and then π_{ab} must be followed. However, the action executed is drawn from the current abstract stochastic policy π_{ab} and we have:

$$P(s'|s, \alpha) = P(s'|s, \alpha, \pi_{ab}) = \frac{P(s' \wedge \alpha | s, \pi_{ab})}{P(\alpha | s, \pi_{ab})} = \frac{P(s' \wedge \alpha | s, \pi_{ab})}{\pi_{ab}(\sigma(s), \alpha)}.$$

Then, we can sample pairs (s', α) from distribution π_{ab} and normalize it with $\pi_{ab}(\sigma(s), \alpha)$. Finally, we can estimate $G_{alternative}^{\pi_{ab}}(\sigma, \alpha)$ by:

$$\hat{G}_{alternative}^{\pi_{ab}}(\sigma, \alpha) = \gamma \frac{\sum_{h=1}^H \sum_{t=0}^{t_h-1} \gamma^t r_{t,h} \sum_{\bar{t}=0}^t \frac{\mathbb{1}_{\{(s,a)|s \in \mathcal{S}_\sigma \wedge a \in \mathcal{A}_\alpha\}}(s_{\bar{t},h}, a_{\bar{t},h})}{\pi_{ab}(\sigma(s), \alpha)}}{|H|}. \quad (4.15)$$

Finally, we estimate $G(\sigma, \alpha)$ by accumulating σ and pairs (σ, α) occurrences with an eligibility trace. This way, we define algorithm AbsProb-RL in Algorithm 12. We call attention to some details of the AbsProb-RL algorithm. First, unlike AbsProb-PI, AbsProb-RL accepts that π_{ab} converges to a deterministic policy, however the policy executed is always ϵ -greedy, this allows that an arbitrary policy can be chosen, mainly if we know a policy resulting from any other process. Second, estimation window $H(i)$ controls how many episodes the agent waits before changing policy π_{ab} and this number can change through time, for example waiting less episodes in the beginning of the learning process and more episodes in the end. Third, function $\mu_G(i)$ allows to control how to combine different periods of executing the same policy; if $\mu_G(i) = 1$, then only local gradient estimation is considered, whereas if $\mu_G(i) = 1/i$, then all local gradients are combined uniformly, finally, if $\mu_G(i) = x$, x being a constant, the exponential smoothing average is considered.

Algorithm 12 AbsProb-RL

```

> Given a task  $\Omega$ 
> Given a step size function  $\delta_\pi$ , learning rate function  $\mu_G(i)$ , discount factor  $\gamma$ , exploration rate  $\epsilon$  and estimation window function  $H(i)$ 
1: Initialize the stochastic abstract policy  $\pi_{ab}$  arbitrarily
2:  $G_{global}(\cdot, \cdot) \leftarrow 0$ 
3: for each iteration  $i$  do
4:    $G_{local}(\cdot, \cdot) \leftarrow 0$ 
5:   for each episode  $h \in \{1, 2, \dots, H(i)\}$  do
6:      $\eta_{ab}(\cdot, \cdot) \leftarrow 0$ 
7:     for each time step  $t \in \{0, 1, 2, \dots\}$  do
8:       observe  $\sigma$ 
9:       choose  $\alpha$  by following  $\pi_{ab}$  with an  $\epsilon$ -greedy strategy
10:      execute  $\alpha$  and observe reward  $r$ 
11:      for each pair  $(\sigma', \alpha') \in \mathcal{S}_{ab} \times \mathcal{A}_{ab}$  where  $\eta_{ab}(\sigma', \alpha') > 0$  do
12:         $G_{local}(\sigma', \alpha') \leftarrow G_{local}(\sigma', \alpha') + \frac{r\gamma^t \eta_{ab}(\sigma', \alpha')}{|H(i)|}$ 
13:         $\eta_{ab}(\sigma, \alpha) \leftarrow \eta_{ab}(\sigma, \alpha) + \frac{1}{(1-\epsilon)\pi_{ab}(\sigma, \alpha) + \frac{1}{\epsilon}}$ 
14:        for each  $\alpha' \in \mathcal{A}_{ab}$  do
15:           $\eta_{ab}(\sigma, \alpha') \leftarrow \eta_{ab}(\sigma, \alpha') - 1$ 
16:      for each pair  $(\sigma', \alpha') \in \mathcal{S}_{ab} \times \mathcal{A}_{ab}$  do
17:         $G_{global}(\sigma', \alpha') \leftarrow (1 - \mu_G(i))G_{global}(\sigma', \alpha') + \mu_G(i)G_{local}(\sigma', \alpha')$ 
18:      for each  $\sigma \in \mathcal{S}_{ab}$  do
19:        find the best direction  $\alpha_\sigma^* = \arg \max_{\alpha \in \mathcal{A}_{ab}} G_{global}(\sigma, \alpha)$ 
20:      for each pair  $(\sigma', \alpha') \in \mathcal{S}_{ab} \times \mathcal{A}_{ab}$  do
21:         $\pi_{ab}(\sigma, \alpha) \leftarrow \begin{cases} (1 - \delta_\pi(i))\pi_{ab}(\sigma, \alpha) + \delta_\pi(i) & , \text{ if } \alpha = \alpha_\sigma^* \\ (1 - \delta_\pi(i))\pi_{ab}(\sigma, \alpha) & , \text{ if } \alpha \neq \alpha_\sigma^* \end{cases}$ 

```

This is the AbsProb-RL algorithm, which uses the Monte Carlo approach to transform AbsProb-PI in a model-free algorithm following an online learning. Section 6.3 presents experiments for parameter tuning of the algorithm and Section 6.5 shows experiments assessing the algorithm effectiveness.

4.3 Multiple source tasks

All methods described up to now tackle one task at a time. We can extend these algorithms so they can solve a set of tasks, instead of just one. Section 3.2.5 shows works on transfer learning that gather knowledge from multiple source tasks. Here we explore two approaches for handling several source tasks:

- *Case-based approach* – Represents the creation of a policy library, from the policies of each source task;
- *Knowledge generalization approach* – Builds a single generalized policy that takes into account all source tasks.

The main idea of the first one is to maintain several pieces of past knowledge that can be useful in future tasks. Then, one of the main concerns lies in the selection, among a collection of past experiences (cases), of the best case(s) to aid the learning in the target task. On the knowledge generalization side, the focus is on how to combine and to represent the solutions of a number of past experiences, extracting their similarities and hence generalizing this previous knowledge. These two approaches are described in the next sections.

4.3.1 Building a policy library

Here we present a slightly modified version of the PLPR algorithm (FERNÁNDEZ; VELOSO, 2006), which is an incremental method to build a policy library from a set of source tasks $\Omega_1, \Omega_2, \dots, \Omega_n$. The library contains a collection of policies, each one being the solution to a source task. However, the library does not necessarily contain all possible policies; to prevent the library from having two policies that are too similar, and thereby, the overhead of selecting between them during the learning process, there must be a criterion to decide whether to include a policy in the library or not. This criterion defines a similarity metric between policies and is a key step for the algorithm.

The algorithm to build the library is given in Algorithm 13. To build the library, the source tasks are tackled one at a time, following a certain order. We use an algorithm to find a policy π_i for each source task Ω_i (line 3). It can be any algorithm that finds an abstract policy for an RMDP, such as AbsProb-RL. When the agent finds this policy it also has to calculate the average gain W_i , as well as the maximum average gain W_{max} within all policies in the library. If the average gain W_i obtained by following this policy π_i is δ_W times greater than the maximum gain obtained in task Ω_i using any of the policies currently in the library; then, policy π_i is added to the library (line 8). W_i is calculated by:

$$W_i = \sum_{s \in \mathcal{S}} b_0(s) \left\{ \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t | \pi_i, s_0 = s \right] \right\}. \quad (4.16)$$

Parameter δ_W controls how permissive to new policies the library is: if $\delta_W = 0$, only one policy (the first) is stored. Conversely, if $\delta_W = 1$, it is most likely that all the learned policies are included.

Algorithm 13 PRPL - algorithm to build a policy library

```

▷ Given an ordered set of source tasks  $\Omega_1, \Omega_2, \dots, \Omega_n$  and a parameter  $\delta_W$ 
1: Initialize a policy library  $L$  with  $L = \{\}$ 
2: for each source task  $\Omega_i$  do
3:   Find a policy  $\pi_i$ 
4:   Obtain  $W_i$ , the average gain obtained following  $\pi_i$  in  $\Omega_i$ 
5:   Obtain  $W_1, W_2, \dots, W_{|L|}$  the average gains obtained following each policy in the
      library
6:    $W_{max} = \max\{W_1, W_2, \dots, W_{|L|}\}$ 
7:   if  $W_{max} < \delta_W W_i$  then
8:      $L = L \cup \{\pi_i\}$ 

```

This algorithm is incremental because for each new task it faces, the policy library is updated. To take full advantage of the algorithm, it is better if the algorithm used to find a policy also uses the policy library. This way, during the process of finding a policy, the average gains of each policy in the library are calculated as they are being used. An architecture for combining AbsProb-RL with the use of a policy library is shown in Chapter 5. Here we presented how to build a policy library. How to use and apply it on a target task is also described in Chapter 5.

4.3.2 Building a generalized policy

In contrast to the construction of the policy library, we propose an alternative method for leveraging past knowledge from multiple source tasks. This method tries to represent common knowledge present on all past experiences. This way, the agent can gather knowledge from several source tasks, given that it has simultaneous access to all of them. The resulting abstract policy is then a *generalized policy*, which comprises solutions to the whole set of tasks. This single policy defines a suboptimal solution for a whole set of source tasks.

We adapt the AbsProb-RL algorithm to this set of source tasks scenario. Our approach is to tackle all source tasks jointly, i.e., alternating tasks on each iteration. By doing this, gradient G_Ω is calculated for each task Ω and then the gradient G_{local} is the sum of all gradients. Thereby, the gradient G_{local} points towards a policy with a value that is the maximum considering the average of all tasks. Algorithm 14 shows the modified version of AbsProb-RL.

We notice that the main differences are in lines 4 and 18. The former is the additional iteration over the set of tasks, which occurs inside the policy iteration loop. The latter is an additional step right after the end of the task loop; it calculates G_{local} by performing the sum of all G_{Ω_j} values for each task Ω_j . Then the algorithm continues the same as Algorithm 12.

It is important that the reward functions of each task are similar to each other, if one desires that all tasks are considered with equal weight. Otherwise, tasks that offer higher rewards will have more importance on the final solution. An example of reward function that all tasks can have is:

$$R(s) = \begin{cases} 1 & \text{if } s \in \mathcal{G}, \\ 0 & \text{otherwise.} \end{cases}$$

To sum up, AbsProb-RL finds a generalized stochastic abstract policy that is a suboptimal solution to the set of source tasks. Actually, this policy is the best abstract policy as if all tasks were merged into a single MDP. We would like to illustrate the use of the two approaches with a simple example, so that we can give some intuition into the use of a method based either on cases or on generalizations.

4.3.3 Illustrative example

Let us consider a simple example: a grid-world where the agent actions are to move one cell north, south, east or west, resulting in its movement unless there is a wall in the desired direction. Besides, the agent cannot see farther than one cell, i.e., it can only perceive adjacent cells, and each cell represents a state. The initial state is always the top-left one and the agent's task is to go to the target location, minimizing the number of steps.

There are two source tasks, illustrated in Figure 4.3. One can notice that each task has 6 ground states. In this environment, there are four types of terrain: dirt, sand, grass and rocks. Different states with the same type of terrain have the same abstract representation and, therefore, are in the same abstract state. There is thus a total of 4 abstract states in each task. As we focus on the transfer of abstract policies, actions are defined per abstract state. In this simple example, we use abstraction only for states, there is no abstraction for action. In other words, the abstract actions are exactly the same as the ground actions. Both tasks are in the same domain class, but as they have different transition functions, they are in different domains (according to our definition in Section 3.1).

In both tasks, the agent always starts in a dirty terrain and has to reach a target in a rocky terrain. We can see that the agent has basically one major decision to make: should it follow the sandy or the grassy path? In Ω_1 , the optimal policy is to follow the sandy path, whereas in Ω_2 the best choice is the grassy one. This is reflected in policies π_1 and π_2 (Figure 4.4), each one solving Ω_1 and Ω_2 , respectively, and both of which are included in the library of policies.

Figure 4.3 – *Source tasks* Ω_1 and Ω_2 – white cells are dirt; light grey, grass; dotted pattern, sand; and dark grey, rocks. Thick black lines represent walls, the initial state is the top-left cell and the target is marked with circles.

Figure 4.4 – *Transferred knowledge* – Optimal policies for the source tasks (π_1 for Ω_1 and π_2 for Ω_2); π_g is a generalized policy that suboptimally satisfies both Ω_1 and Ω_2 . Actually, the policies are stochastic, but the arrows just represent the most likely action of each state.

If we follow the generalization approach, instead of having a library with 2 policies, a single generalized policy π_g is built. The intuition is that, while each task has a different optimal path, in both tasks the grassy path can effectively lead the agent to the goal. The generalized policy thus assigns a higher probability for following the grassy path that, despite not being optimal for every source task, is the best global solution (following the sandy path in Ω_2 would result in a dead-end). In this example, coincidentally π_g is equal to π_2 , considering just the most likely actions.

Now consider a target task Ω_3 , shown in Figure 4.5a, in which the agent is learning with reinforcement learning. It is similar to the source tasks, but it is a new and unseen task. Policies π_1 , π_2 and π_g are used to guide its exploration in the environment, expecting it to be faster than simply using random actions. Further details on how the policies are applied to improve performance in reinforcement learning are in Section 5.2. If the agent uses the policy library, it can choose either π_1 or π_2 to help it. When the agent chooses π_1 , the guidance provided by the policy is extremely helpful, as the most probable action

for the dirty and sandy terrains is ‘go right’, which is the same as the optimal policy. When the agent chooses π_2 , it provides some help, as it does lead the agent to the goal, but it leads through a longer way than π_1 (through the grassy path). If the agent uses the generalized policy, it does not have to perform any policy selection, it just uses π_g to accelerate its learning as much as π_2 would do. The purpose of using these past policies is preventing the agent to struggle to find a way at the beginning of the reinforcement learning process. With their help, the agent shows good behavior from the beginning and, as learning progresses, it eventually finds the optimal policy for that task.

Figure 4.5 – *Target tasks Ω_3 and Ω_4* – Tasks to be learned with the aid of past policies (π_1 and π_2 , or π_g). Arrows indicate the optimal action for each state.

Let us consider another target task, Ω_4 , shown in Figure 4.5b. In this case, π_2 and π_g are the ones that provide the best help. π_1 , however, is not very helpful; actually, it leads the agent to a dead-end.

This example is designed to illustrate the intuition behind the choice of building either a generalized policy or a library. When using a library of policies, some policies might be helpful while others might not. It is important for the agent to rapidly realize this during the learning process of a target task to avoid *negative transfer* (TORREY; SHAVLIK, 2009), i.e., to avoid decreasing the performance of the agent when compared to one without any transfer. The generalized policy approach is a solution to this issue, even if it may not always be the most effective guide. Nevertheless, our experiments show that, when looking for better results on average, the generalized approach performs better. Section 6.5 covers this comparison in detail.

Algorithm 14 AbsProb-RL - multiple source tasks

▷ Given a set of tasks $\Omega_1, \Omega_2, \dots, \Omega_n$
 ▷ Given a step size function δ_π , learning rate function $\mu(i)$, discount factor γ , exploration rate ϵ and estimation size function $H(i)$

- 1: Initialize the stochastic abstract policy π_{ab} arbitrarily
- 2: $G_{global}(\cdot, \cdot) \leftarrow 0$
- 3: **for** each iteration i **do**
- 4:   **for** each task Ω_j **do**
- 5:   **for** each episode $h \in \{1, 2, \dots, H(i)\}$ **do**
- 6: $G_{\Omega_j}(\cdot, \cdot) \leftarrow 0$
- 7: $\eta_{ab}(\cdot, \cdot) \leftarrow 0$
- 8:   **for** each time step $t \in \{0, 1, 2, \dots\}$ **do**
- 9:   observe σ
- 10:   choose α by following π_{ab} with an ϵ -greedy strategy
- 11:   execute α and observe reward r
- 12:   **for** each pair $(\sigma', \alpha') \in \mathcal{S}_{ab} \times \mathcal{A}_{ab}$ where $\eta_{ab}(\sigma', \alpha') > 0$ **do**
- 13: $G_{\Omega_j}(\sigma', \alpha') \leftarrow G_{\Omega_j}(\sigma', \alpha') + \frac{r\gamma^t \eta_{ab}(\sigma', \alpha')}{|H(i)|}$
- 14: $\eta_{ab}(\sigma, \alpha) \leftarrow \eta_{ab}(\sigma, \alpha) + \frac{1}{(1-\epsilon)\pi_{ab}(\sigma, \alpha) + \frac{1}{\epsilon}}$
- 15:   **for** each $\alpha' \in \mathcal{A}_{ab}$ **do**
- 16: $\eta_{ab}(\sigma, \alpha') \leftarrow \eta_{ab}(\sigma, \alpha') - 1$
- 17:   **for** each pair $(\sigma', \alpha') \in \mathcal{S}_{ab} \times \mathcal{A}_{ab}$ **do**
- 18: $G_{local}(\sigma', \alpha') = \sum_{j=1}^n G_{\Omega_j}(\sigma', \alpha')$
- 19:   **for** each pair $(\sigma', \alpha') \in \mathcal{S}_{ab} \times \mathcal{A}_{ab}$ **do**
- 20: $G_{global}(\sigma', \alpha') \leftarrow (1 - \mu(i))G_{global}(\sigma', \alpha') + \mu(i)G_{local}(\sigma', \alpha')$
- 21:   **for** each $\sigma \in \mathcal{S}_{ab}$ **do**
- 22:   find the best direction $\alpha_\sigma^* = \arg \max_{\alpha \in \mathcal{A}_{ab}} G_{global}(\sigma, \alpha)$
- 23:   **for** each pair $(\sigma', \alpha') \in \mathcal{S}_{ab} \times \mathcal{A}_{ab}$ **do**
- 24:
$$\pi_{ab}(\sigma, \alpha) \leftarrow \begin{cases} (1 - \delta_\pi(i))\pi_{ab}(\sigma, \alpha) + \delta_\pi(i) & , \text{ if } \alpha = \alpha_\sigma^* \\ (1 - \delta_\pi(i))\pi_{ab}(\sigma, \alpha) & , \text{ if } \alpha \neq \alpha_\sigma^* \end{cases}$$

---

5 LEARNING AGENT ARCHITECTURE

In this chapter we present our proposal to achieve transfer learning: a 2-layered architecture for the agent, exploring abstraction and generalization and having abstract policies as transferred knowledge. First, Section 5.1 presents an overview of the architecture, its components and functions. Section 5.2 presents ψ -reuse, an exploration strategy that incorporates the use of a past abstract policy into the action selection process. Finally, Section 5.3 shows the proposed architecture, namely S2L-RL: Simultaneous Two-Layer Reinforcement Learning, in detail.

5.1 Overview

As we have already noted, we are primarily interested in leveraging past knowledge obtained from source tasks to improve performance of the agent on a new target task. We achieve this by exploring state and actions abstractions and thereby generalizing knowledge. We propose an agent architecture which enables the agent to use abstract policies to guide it in a new task, as well as build these abstract policies for future tasks. Last chapter we presented how abstract policies can be built and this architecture answers how they can be used to aid the agent in the learning process.

Figure 5.1 summarizes the architecture. Its main characteristic is that it has two *layers* (or levels): the abstract level and the ground level. Just the ground level interacting with the environment would be the ‘classic’ reinforcement learning agent. Here, the agent learns simultaneously two policies, one at each level; an abstract policy at the abstract level and a ground policy at the ground level. In addition, the agent can use one or more past abstract policies to help it solve the current problem. The arrow from the abstract policy to the past policies indicates that this abstract policy learned can be included in the set of past ones for a future task.

Interactions with environment clearly occurs always at the ground level, but the experiences $\langle s_t, a_t, r_t, s_{t+1} \rangle$ obtained at this level are used in both, abstract and ground, so that values can be updated in both levels. Therefore it needs to pass through the abstraction functions ϕ_s and ϕ_a to be mapped to $\langle \sigma_t, \alpha_t, r_t, \sigma_{t+1} \rangle$ before being used at the abstract level. Similarly, when an abstract policy points an abstract action to be executed, it has first to pass through the grounding procedure.

Learning at the abstract level serves two purposes:

1. Provide guidance in the current task;

Figure 5.1 – The two-layered learning agent architecture

2. Prepare a policy for being used in future tasks.

Because the abstract state space is smaller than the ground state space, learning can be faster at the abstract level than at the ground level, as we show in Figure 4.1. The abstract policy can be used to improve the agent's performance at the initial episodes, in which it is usually very poor due to the lack of information. When the agent stops learning the current task, besides possessing the optimal ground policy for the current task it also produces an abstract policy for it. As a result, the agent can keep this policy to use on future tasks in a policy library for example.

Example 7 Assume the agent has two tasks to learn: first Ω_{source} and after Ω_{target} . For solving the first task Ω_{source} , the agent does not have any past policy. Then, with this 2-layered architecture, it will learn a ground policy π_{gnd}^{source} and an abstract policy π_{abs}^{source} for task Ω_{source} . As the abstract policy is learned faster than the ground, π_{abs}^{source} is used to speed-up learning of π_{gnd}^{source} . This way, π_{gnd}^{source} is learned faster than using a plain ground learner. After several episodes, agent has learned nearly optimal π_{gnd}^{source} and π_{abs}^{source} policies. For solving task Ω_{target} , agent now has one past policy, π_{abs}^{source} to be transferred and reused. Then, it will learn a ground policy π_{gnd}^{target} and an abstract policy π_{abs}^{target} for task Ω_{target} . In the beginning of the learning process, agent uses π_{abs}^{source} more often, guiding it to find the solution for the current task. Then, as learning progresses, π_{gnd}^{target} and π_{abs}^{target} are being refined and tuned for Ω_{target} and the use of the past policy π_{abs}^{source} decreases. At an intermediate

phase of learning, π_{abs}^{target} can guide learning like in the previous task and eventually π_{gnd}^{target} is refined enough that no abstract policy can provide any useful guidance.

The objective of the architecture is that resulting performance over time of the agent when learning Ω_{target} is better than the performance of an agent without transfer and also of an purely ground agent.

We thus have a framework where the past policies give an initial jumpstart to the ground level; then the abstract level is quickly refined and continues to provide guidance to the ground level; finally the ground level is finely tuned to the target problem, abandoning the guidance of the abstract level. This framework is an extension of published work (KOGA et al., 2013). We now turn these intuitions into reality and in the next sections we show how the framework works in practice.

5.2 ψ -reuse exploration strategy

To achieve the acceleration of the learning process through policy transfer, one can use the knowledge they convey to bias the exploration process, which is usually purely random. During the reinforcement learning process, the agent must balance the exploration of the environment and the exploitation of both the past policy and the policy being learned. In a classic exploration strategy, agent balances between two choices: exploitation of current policy $\pi_{current}$ and random exploration. An agent that follows the ϵ -greedy strategy, for example, explores with probability ϵ and exploits with $1 - \epsilon$. We present the ψ -reuse exploration strategy, which balances among three choices: exploitation of current policy $\pi_{current}$, random exploration and exploitation of a past policy. Figure 5.2 illustrates the differences between the strategies.

Figure 5.2 – *Exploration strategies* – Apart from exploring with π_{random} and exploiting $\pi_{current}$, ψ -reuse adds one more policy to be exploited, π_{past} .

The ψ -reuse exploration strategy meets this need for balancing exploration and exploitation with the aid of parameter ψ ($0 \leq \psi \leq 1$). It combines ideas of other reuse strategies (FERNÁNDEZ; GARCÍA; VELOSO, 2010; MATOS et al., 2011a), modifying them for stochastic abstract policies, and it is detailed in Algorithm 15. By the ψ -reuse exploration strategy, the agent will then follow the given abstract past policy π_{past} with a probability ψ , the policy being learned $\pi_{current}$ with a probability $(1 - \psi)(1 - \epsilon)$ and the random policy π_{random} with probability $(1 - \psi)\epsilon$. This is equivalent to follow an ϵ -greedy strategy (Algorithm 2) with probability $1 - \psi$ and past policy reuse with probability ψ . Random exploration is always necessary for guaranteeing a complete exploration of the state space.

Algorithm 15 ψ -reuse($Q, s, \pi_{ab}, \epsilon, \psi$)

- ▷ Given action-value function Q , current state s , a stochastic abstract policy π_{ab} , exploration rate ϵ and reuse rate ψ
  - ▷ Returns chosen action
  - 1: **if** random($[0, 1]$) $\leq \psi$ **then** ▷ uniform distribution over $[0, 1]$
  - 2: $a \leftarrow \text{grounding}(\pi_{ab}, s)$ ▷ uses π_{ab} w.p. ψ .
  - 3: **else**
  - 4: $a \leftarrow \epsilon$ -greedy(Q, s, ϵ)
  - 5: **Return** a
-

This strategy shows how an abstract policy can be used to guide the ground level interactions, defining the behavior of the agent when interacting with the environment. ψ -reuse is an important part of the architecture, which is further detailed in the next section.

5.3 Simultaneous Two-layer Reinforcement Learning

The proposed learning agent architecture has to deal with a number of policies when solving a task $\Omega = \langle \Sigma, \mathcal{B}, \mathcal{S}, \mathcal{A}, T, R, \mathcal{G}, b^0 \rangle$. They are:

- A deterministic policy being learned at the ground level, π_{gnd} , or π_0 ;
- An abstract policy being learned at the abstract level, π_{abs} , or π_1 ;
- n_p abstract policies from previous source tasks, π_{past}^i , or $\pi_i, i = 2, \dots, n_p + 1$.

Several RL algorithms, such as Q-Learning and Sarsa(λ), can be used at the ground level to build π_{gnd} . These two are amongst the most popular RL algorithms and are the ones we use in this work. For the abstract level we presented two possible algorithms:

AbsSarsa(λ) (Algorithm 10) and AbsProb-RL (Algorithm 12). Both of them can be used at the top level in order to build π_{abs} .

In our framework learning proceeds in multiple episodes. Each episode positions the agent in an initial state according to b_0 and terminates when either the goal state s_g is reached or a maximum number of steps is achieved. Here we consider that each task is defined by only one goal state, i.e. in each task $\mathcal{G} = \{s_g\}$.

At the beginning of each episode, the agent selects among possible policies to apply: π_{gnd} , π_{abs} and the (if existing) π_{past} policies learned previously in similar tasks. Which policy from all available will the agent use? We use the same metric as the PRPL algorithm used for building a policy library (refer to Section 4.3.1) to compare policies. An estimate of the policy value is used as a similarity metric to compare all choices. This idea has been explored in PRQ-Learning (FERNÁNDEZ; GARCÍA; VELOSO, 2010), which implements a probabilistic policy reuse to accelerate Q-Learning.

This policy selection is guided by a probabilistic choice, according to values $W^{\pi_{gnd}}$, $W^{\pi_{abs}}$ and n_p values of $W^{\pi_{past}}$. The values W_k^π at any episode k are the average reinforcement received per episode after executing that policy, i.e.,

$$W_k^\pi = \frac{1}{\sum_{j=1}^{k-1} \mathbf{1}(\pi_{chosen}^j = \pi)} \sum_{j=1}^{k-1} \sum_{t=1}^{t_{max}} \mathbf{1}(\pi_{chosen}^j = \pi) \gamma^t r_{k,t}, \quad (5.1)$$

where the function $\mathbf{1}(\pi_{chosen}^j = \pi)$ indicates if the policy π was chosen in the episode j , t_{max} is the maximum number of steps in each episode, γ is the discount factor and $r_{k,t}$ is the reward received in episode k at time step t .

For selecting a policy we use a *softmax* selection, in which all policies are ranked and weighted according to their value estimates W^π . The higher the W^π value, the higher the probability of π being chosen. The most common softmax method uses a Boltzmann distribution (SUTTON; BARTO, 1998) and chooses policy π with probability

$$\frac{e^{\tau W^\pi}}{\sum_{i=0}^{np+1} e^{\tau W^{\pi_i}}} \quad (5.2)$$

where W^{π_j} , $j = 0, \dots, np+1$ represents the W value of each of the policies considered and τ , $\tau \geq 0$, is a temperature parameter. When $\tau = 0$, all policies are equiprobable. Higher temperatures indicate a more greedy selection. The τ parameter increments by $\Delta\tau$ after each episode, which means that the agent becomes greedier along time if $\Delta\tau > 0$.

If the policy π_{chosen}^k selected is π_{gnd} , it will be applied for an entire episode k following a ψ -reuse strategy with $\psi = 0$, i.e., without any reuse. Actually, ψ being 0 means that it follows an ϵ -greedy episode. If the policy π_{chosen}^k selected is π_{abs} or any past policy, then it will be the policy used to guide the agent during episode k , following an ψ -reuse

strategy with $\psi = \psi_0$. During an episode, ψ decays exponentially by an amount given by parameter ν , which means that the guidance is more often at the beginning of the episode.

At the end of each episode k , the average reinforcement W received in that episode is calculated and the corresponding W^π is updated, i.e., we update $W^{\pi_{gnd}}$, $W^{\pi_{abs}}$, or one of the past policies $W^{\pi_{past}}$. As this amount of reinforcement was obtained with the aid of the reuse of only one policy, agent can estimate the value of each policy and thereby use the W^π values as a similarity metric among all policies.

Probabilities of each policy being selected change over time because π_{gnd} and π_{abs} are being built and thus their W^π values are changing, and also because τ increases. When the agent starts learning, π_{gnd} is equal to π_{random} , since the Q function is usually initialized with the same value to all state-action pairs. If we assume that some π_{past} was learned considering a similar task, the actions the agent will consider have the property that they used to be good to solve a similar problem before. Therefore the agent should be able to obtain a better initial performance, i.e. a higher value of R_t in the first episodes. This policy selection allows the agent to shift from using past policies in the beginning, because they have higher W values, to using π_{abs} when it achieves a better value than the past policies, to finally using only π_{gnd} once it is tuned and nearly optimal for the target task.

Combining the ideas of the ψ -reuse exploration strategy for policy reuse, the two levels with simultaneous learning and the probabilistic policy selection of PRQ-Learning, we propose an agent architecture called Simultaneous Two-layer Reinforcement Learning, or S2L-RL. It is described in Algorithm 16, where:

- K is the number of episodes to be executed
- t_{max} is the maximum number of steps at each episode
- ϵ , $0 \leq \epsilon \leq 1$, is the exploration probability for the ϵ -greedy strategy;
- γ , $0 \leq \gamma < 1$, is the discount factor;
- μ , $0 \leq \mu \leq 1$, is the learning rate;
- ψ_0 , $0 \leq \psi_0 \leq 1$, is the initial reuse probability per episode;
- ν , $0 \leq \nu \leq 1$, is the decay rate of the reuse probability;
- τ is the temperature parameter for the softmax policy selection;
- $\Delta\tau$ is the increment size for τ ;
- W_0 is the estimated value of the policy at the ground level, $W^{\pi_{gnd}}$;

- W_1 is the estimated value of the policy at the abstract level, $W^{\pi_{abs}}$;
- $W_i, i = 2 \dots (np + 1)$, is the estimated value of each past policy $W^{\pi_{past}^i}$;
- U_0 is the number of episodes π_{gnd} was selected;
- U_1 is the number of episodes π_{abs} was selected and
- $U_i, i = 2 \dots (n_p + 1)$, is the number of episodes each past policy π_{past}^i was selected.

Here we are describing S2L-RL with Q-Learning at the ground level and AbsProb-RL at the abstract level. It could be easily adapted for being used with Sarsa(λ) or AbsSarsa(λ), with a few modifications on the update equations and attention on update restrictions of each algorithm.

For clarity, we left implicit the update steps at the abstract level (lines 20 and 25). The abstract level builds π_{abs} following the AbsProb-RL algorithm, which is detailed in Section 4.2.3. Line 20 uses the experience $\langle \sigma, \alpha, r, \sigma' \rangle$ to update eligibility trace $e(\sigma, \alpha)$ and gradient G_{local} values. Then at the end of each episode, on line 25, AbsProb-RL may perform a policy improvement step depending on the estimation window size H .

To sum up, the behavior of the agent with the S2L-RL architecture is to learn by using policies π_{past} or π_{abs} to guide exploration and gradually replace it with the policy in the ground level π_{gnd} , while using π_{random} just as to guarantee exploration and convergence to optimality. The choice is driven by the estimated value of each policy to the learning task. Next chapter shows experiments with this architecture.

Algorithm 16 S2L-RL - Simultaneous Two-layer Reinforcement Learning

---

▷ Given the task Ω to be solved and n_p past policies ($n_p = 0$ means no past policy)
 ▷ Given parameters $K, t_{max}, \epsilon, \gamma, \mu, \psi_0, \nu, \tau_0, \Delta\tau$
 1: Initialize $Q(s, a)$ and $\pi_{gnd}(s)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$ ▷ Ground level
 2: Initialize $\pi_{abs}(\sigma, \alpha)$ arbitrarily, for all $\sigma \in \mathcal{S}_{ab}, \alpha \in \mathcal{A}_{ab}$ ▷ Abstract level
 3: ▷ All policies are indexed as $\pi_i, i = 0, \dots, n_p + 1$. Index $i = 0$ represents π_{gnd} , $i = 1, \dots, n_p$, and $i = n_p + 1$, the past policies
 4: $W_i \leftarrow 0$, for $i = 0, \dots, n_p + 1$ ▷ Average reward received by using π_i
 5: $U_i \leftarrow 0$, for $i = 0, \dots, n_p + 1$ ▷ Number of episodes π_i has been chosen
 6: $\tau \leftarrow \tau_0$
 7: **for** Each episode $k = 1$ to K **do**
 8:    $P[i] \leftarrow \frac{e^{\tau W_i}}{\sum_{j=1}^{n_p+1} e^{\tau W_j}}$, for $i = 1, \dots, n_p + 1$ ▷ Probability of π_i to be chosen
 9:   Choose index $c \in \{1, \dots, n_p + 1\}$ according to P
 10:   **if** $\pi_c = \pi_{gnd}$ **then**
 11:      $\psi \leftarrow 0$ ▷ If the selected policy is the ground one, there is no reuse.
 12:   **else**
 13:      $\psi \leftarrow \psi_0$
 14:    $W \leftarrow 0; t \leftarrow 0$
 15:   Obtain the initial state s from b^0
 16:   **while** $t < t_{max}$ and $s \notin \mathcal{G}$ **do**
 17:      $a \leftarrow \psi\text{-reuse}(Q, s, \pi_c, \epsilon, \psi)$
 18:     Execute a , observe new state s' and reward r
 19:      $Q(s, a) \leftarrow (1 - \mu)Q(s, a) + \mu[r + \gamma \max_{a'} Q(s', a')]$
 20:     Use $\langle \sigma = \phi_s(s), \alpha = \phi_a(a), r, \sigma' = \phi_s(s') \rangle$ to update $e(\sigma, \alpha)$ and $G_{local}(\sigma, \alpha)$ of the abstract level
 21:      $W \leftarrow W + \gamma^t r$
 22:      $\psi \leftarrow \nu \psi$
 23:      $t \leftarrow t + 1; s \leftarrow s'$
 24:    $\pi_{gnd}(s) \leftarrow \arg \max_a Q(s, a), \forall s \in \mathcal{S}$
 25:   Update π_{abs} according to AbsProb-RL
 26:    $W_c \leftarrow \frac{W_c U_c + W}{U_c + 1}$
 27:    $U_c \leftarrow U_c + 1$
 28:    $\tau \leftarrow \tau + \Delta\tau$
 29: **Return** π_{gnd}, π_{abs}

---

6 EXPERIMENTS AND RESULTS

This chapter presents the experiments executed to evaluate the performance of an RL agent using the S2L-RL architecture with AbsSarsa(λ) or AbsProb-RL. All experiments were conducted in a simulated environment. Section 6.1 describes the robotic navigation domain class used and its relational representation; Section 6.2 shows the experimental setup used in this work; Section 6.3 presents an analysis on different parameters values of AbsProb-RL; Section 6.4 presents experiments with an agent using the S2L-RL architecture with AbsSarsa(λ) at the abstract level and finally Section 6.5 presents experiments using S2L-RL with AbsProb-RL at the abstract level and also performs comparisons between transfer learning methods.

6.1 Navigation Domain Class

Let us describe the domain class used in our experiments. It is a grid-like navigation domain class, a common environment in reinforcement learning works (SUTTON, 1990; HAUSKRECHT et al., 1998; FERNÁNDEZ; VELOSO, 2006; DEVLIN; KUDENKO, 2012). Here we extend it with a relational representation.

There is an indoor environment which contains two kinds of places: rooms and corridors, with doors connecting them. There are several markers in these places for identification (room markers, corridor markers) and these markers, as well the doors, are used as objects in the relational representation. A robot (our agent) can navigate through the rooms and corridors and the space is discretized in cells, each cell possessing a marker and corresponding to a different state. A task always defines a single location the robot must reach, which is always a cell inside a room. Thus, there is one goal state for each task and the task is finished when the robot reaches it. One example can be seen in Figure 6.1, with the goal state in cell number 4.

Our concern is to control the robot on a higher level of abstraction, so that the robot can sense doors and identify whether it is inside a room or a corridor. Additionally, the robot also has a positioning system, making it able to tell whether it is near or far from the goal location and also if some object is closer or farther from the goal than the robot itself. We describe each ground state using the vocabulary $C \cup P_L \cup P_O \cup P_P$, detailed below.

C is the set of objects: rooms markers, corridor markers and doors. Room markers are objects inside a room, that identify the room. Analogously, corridor markers are objects

Figure 6.1 – *Navigation Domain $\mathcal{D}_1$* – Thick lines represent walls; darker cells, rooms; and white cells, corridors. Discontinuities in the walls represent doors. Each numbered cell represents a ground state. Cell number 4 is the goal, as an example; the states considered near the goal are also marked in darker colors.

inside corridors. Doors are passages from one room to another or to a corridor. At any position in the environment there is always a marker the robot believes it is (hence it can never be inside a room and in a corridor at the same time). This also means that the resolution of markers in the environment is a limiting factor for the resolution of the state space. Some experiments were made with as many markers as cells as in domain $\mathcal{D}_3$ shown in Figure 6.8, to be comparable to other works that used the same domain. Conversely, other experiments had just one marker per room, so, marker-wise, each room has only one state (other objects such as doors can create more than one state inside the same room).

The robot has a set of sensors that can sense a marker at a distance of one cell, while it may sense doors that are up to two cells away. All state predicates are divided in three sets: P_L , P_O , and P_D , all of which can be seen in Table 6.1. P_L contains predicates related to the location of the agent; P_O contains predicates related to observations and P_P contains predicates related to the relative position of the objects. Here we describe them all.

- P_L :
  - $\text{inRoom}(\mathbf{R})$ indicates that the agent is inside a room and the closest marker is room marker $\mathbf{R}$;

- `inCorridor(C)` indicates that the agent is in a corridor and the closest marker is corridor marker `C`.
- P_O :
  - `seeDoor(D)` means that the agent can see a door, but it is at least two cells distant (as the robot cannot see doors farther than two cells away, this predicate is valid only when the robot is exactly two cells away from a door);
  - `seeAdjRoom(R)` indicates that the agent is close to a door (one cell away) and can even see that a room (with room marker `R`) lies through it;
  - `seeAdjCorridor(C)` indicates that the agent is close to a door (one cell away) and can even see that a corridor (with corridor marker `C`) lies through it;
  - `seeEmptySpace(M)` means that the agent sees a free space close to a room or corridor marker `M`, where it could move to;
  - `nearGoal` is true if the agent is at a close distance to the goal, i.e., at a Manhattan distance to the goal lower than 5.
- P_P :
  - `closeGoal(X)` indicates that object `X` is closer than the agent to the goal;
  - `farGoal(X)` indicates that object `X` is farther than the agent to the goal.

Table 6.1 – State predicates used in navigation domain class, where `R` is a room marker, `C` is a corridor marker, `M` is either a room or corridor marker, `D` is a door and `X` is any object.

State predicates		
P_L	P_O	P_P
<code>inRoom(R)</code>	<code>seeDoor(D)</code>	<code>closeGoal(X)</code>
<code>inCorridor(C)</code>	<code>seeAdjRoom(R)</code>	<code>farGoal(X)</code>
	<code>seeAdjCorridor(C)</code>	
	<code>seeEmptySpace(M)</code>	
	<code>nearGoal</code>	

Background knowledge $\mathcal{B}$ defines rules and restrictions to the state description. A state must always contains one, and only one, predicate from P_L , which means that the agent is whether inside a room or a corridor, never at both or none. For each predicate from P_O in the state description, there must be always one from P_P , both with the same object, except if the predicate has zero arity (`nearGoal`).

For example, one possible ground state is:

$$s_1 = \text{inRoom}(r_{11}) \wedge \text{seeDoor}(d_1) \wedge \text{closeGoal}(d_1) \wedge \text{seeEmptySpace}(r_{12}) \\ \wedge \text{closeGoal}(r_{12}) \wedge \text{seeEmptySpace}(r_{13}) \wedge \text{farGoal}(r_{13}) \wedge \text{nearGoal}.$$

The agent is inside a room and at a close distance to the goal. The closest marker is r_{11} . $\text{seeDoor}(d_1) \wedge \text{closeGoal}(d_1)$ means that the agent is seeing a door d_1 that is closer to the goal than the agent is. It also sees two other markers in the room, r_{12} and r_{13} , one closer to the goal and the other farther to the goal, respectively.

Regarding actions, they are described by the set of predicates P_A and can be seen in Table 6.2. All action predicates have one of two suffixes: $\text{AppGoal}(X)$ or $\text{AwayGoal}(X)$, that indicate whether the agent is going to a direction approaching the goal or moving away from the goal when going towards object X , respectively. Actions ending with $\text{AppGoal}(X)$ are only available in states with $\text{closeGoal}(X)$. Likewise, actions ending with $\text{AwayGoal}(X)$ are only available in states with $\text{farGoal}(X)$. Remember that all actions are atoms, i.e., they are formed by just one predicate. The available actions are:

- $\text{goToDoor}[\text{App}|\text{Away}]\text{Goal}(D)$ means that the agent must go closer to door D ; this action is available in states with $\text{seeDoor}(D)$ and appropriate element from P_P .
- $\text{goToRoom}[\text{App}|\text{Away}]\text{Goal}(R)$ means that the agent must enter room with room marker R ; this action is available in states with $\text{seeAdjRoom}(D)$ and appropriate element from P_P .
- $\text{goToCorridor}[\text{App}|\text{Away}]\text{Goal}(C)$ means that the agent must enter corridor with corridor marker C ; this action is available in states with $\text{seeAdjCorridor}(D)$ and appropriate element from P_P .
- $\text{goToEmpty}[\text{App}|\text{Away}]\text{Goal}(M)$ means that the agent must go to an area where the closest marker will be M ; this action is available in states with $\text{seeEmptySpace}(M)$ and appropriate element from P_P .

Table 6.2 – Set of action predicates P_A used in navigation domain class, where R is a room marker, C is a corridor marker, M is either a room or corridor marker and D is a door.

Action predicates	
$\text{goToDoorAppGoal}(D)$	$\text{goToDoorAwayGoal}(D)$
$\text{goToRoomAppGoal}(R)$	$\text{goToRoomAwayGoal}(R)$
$\text{goToCorridorAppGoal}(C)$	$\text{goToCorridorAwayGoal}(C)$
$\text{goToEmptyAppGoal}(M)$	$\text{goToEmptyAwayGoal}(M)$

Not all actions are available at all states, e.g. `goToDoorAppGoal(D)` is only available at states with `seeDoor(D) ∧ closeGoal(D)` in their description. These suffixes in actions may seem redundant information but they play an important role at the abstract level. This means that, for example, at the abstract level, the agent could decide between going to a room that is closer to the goal (`goToRoomAppGoal(X)`) or going to a corridor that is farther (`goToCorridorAwayGoal(Y)`). Moreover, the ground level is not affected by this extra information; at any ground state there are at most four available ground actions, given the nature of each state.

Abstraction function ϕ_s is defined by simply substituting constants to variables in the conjunctions that represent a ground state. If the substitution produces two or more equivalent pairs of predicates $p_1 \wedge p_2, p_1 \in P_O, p_2 \in P_P$, then only one pair remains. Analogously, abstraction function ϕ_a also is defined by simply substituting constants to variables in the atoms that represent a ground action. For example, the abstraction of state s_1 defined above is:

$$\begin{aligned}\sigma_1 = \phi_s(s_1) = & \text{inRoom}(W) \wedge \text{seeDoor}(X) \wedge \text{closeGoal}(X) \wedge \text{seeEmptySpace}(Y) \\ & \wedge \text{closeGoal}(Y) \wedge \text{seeEmptySpace}(Z) \wedge \text{farGoal}(Z) \wedge \text{nearGoal}.\end{aligned}$$

Another example:

$$\begin{aligned}s_2 = & \text{inCorridor}(c_2) \wedge \text{seeDoor}(d_2) \wedge \text{farGoal}(d_2) \wedge \text{seeDoor}(d_3) \\ & \wedge \text{farGoal}(d_3) \wedge \text{seeEmptySpace}(c_1) \wedge \text{closeGoal}(c_1) \\ & \wedge \text{seeEmptySpace}(c_3) \wedge \text{farGoal}(c_3). \\ \sigma_2 = \phi_s(s_2) = & \text{inCorridor}(W) \wedge \text{seeDoor}(X) \wedge \text{farGoal}(X) \wedge \text{seeEmptySpace}(Y) \\ & \wedge \text{closeGoal}(Y) \wedge \text{seeEmptySpace}(Z) \wedge \text{farGoal}(Z).\end{aligned}$$

Here we note that ground state s_2 sees two different doors, d_2 and d_3 . The abstract state σ_2 , however, loses this information, containing simply `seeDoor(X) ∧ farGoal(X)` in its description. This could be translated to “the agent can see *at least* one door farther to the goal than itself”.

Transitions probabilities are higher than 0 between two adjacent states (two adjacent cells). There is always an error probability of transiting between states of 0.05, in which case the agent does not change state. In other words, after executing action a in state s , the agent transits to expected next state s' with probability 0.95 and stays in state s with probability 0.05.

For each task, the reward function is fairly simple. Goal states give positive rewards and all others have zero reward value:

$$R(s) = \begin{cases} 1 & \text{if } s \in \mathcal{G}, \\ 0 & \text{otherwise.} \end{cases}$$

Each task has only one goal state s_g , therefore $\mathcal{G} = \{s_g\}$. This goal state also has the property of always being a state inside a room. The agent tasks are always to arrive to a room location. Initial state probability distribution b^0 is uniform for all states.

6.2 Experimental Setup

We provide here information about all hardware and software used in the experiments of this work. Hardware used was a computer with Intel® Core™ i5-2400 CPU @ 3.10 GHz processor, 8 GB of RAM with Ubuntu 12.04 LTS operating system. Software used was MATLAB® R2009a.

All algorithms implementation and source codes are available under the GPLv2 license (GNU General Public License version 2) at <https://github.com/MLK3/ia>.

6.3 Parameter analysis in AbsProb-RL

Before executing any experiment with the whole S2L-RL architecture, we perform some experiments to analyze the influence on the performance of algorithm AbsProb-RL (Algorithm 12) the parameters $\mu_G(i)$ (learning rate, line 17) and $H(i)$ (estimation window function, on line 5) cause, parameters whose values may vary according to the current iteration i ,

The experiments were made in navigation domain $\mathcal{D}_2$ (Figure 6.2) and the task is to reach goal state 16 (inside room with room marker $\mathbf{r}_6$). Only learning at the abstract level is considered, i.e., we simply run AbsProb-RL to learn this task, not the full S2L-RL architecture. We ran a total of $K = 10000$ episodes, each with maximum number of steps $t_{max} = 500$. Parameters used: fixed step size $\delta_\pi = 0.05$, $\epsilon = 0.05$ and $\gamma = 0.9$. We evaluate the value $W^{\pi_{abs}}$ of the resulting abstract policy π_{abs} according to Equation 4.16.

In the first experiment, we use learning rate $\mu_G(i) = 1$. This means that G_{global} is the average of the $H(i)$ values of G_{local} of each iteration i and it is recalculated every iteration. This choice of μ is because the gradient value depends on the current policy, therefore each time this policy is changed (i.e., each iteration) the estimate must be redone. Given this learning rate, we compare three alternatives to $H(i)$. Estimation window $H(i) = 10i$ indicates that the policy is improved every n episodes and this number n increases by 10

Figure 6.2 – Domain $\mathcal{D}_2$ – Thick lines represent walls and doors are marked with dn , $n \in \{1, \dots, 11\}$. Each numbered cell represents a ground state. There is only one room marker per room. They are represented by rm , $m \in \{1, \dots, 11\}$. Corridor markers are omitted.

every iteration. That is, the first policy improvement occurs after 10 episodes, the second after more 20 (new) episodes, and so on. This number increases so the agent can better estimate gradient values in the course of time, because as time passes it becomes harder to find the best direction to improve the policy. We tested $H(i) = 20i$ and $H(i) = 50i$ as well.

Figure 6.3 illustrates the results of the first experiment. It shows the mean policy value of 10 executions. We can notice that smaller increment sizes present a lower asymptote after a certain amount of episodes, despite a slightly faster learning (higher slope) in the beginning. $H(i) = 50i$ has a slower beginning than $H(i) = 10i$, but after 4000 episodes its policy value increases faster. There is a balance between large windows that may be slow in the beginning and short windows that may not be effective in the end. $H(i) = 20i$ presents the best trade-off in this example. Nonetheless, regardless of $H(i)$ values, AbsProb-RL performance did not achieve desirable results since none of the curves reached the optimal policy value $W^{\pi^*} = 0.546$.

To try to improve performance of AbsProb-RL, the second experiment investigates different forms to the learning rate μ_G . The first form is the one used in the last experiment, the *average per iteration*. An alternative that has proved to be more efficient is the *global average*. In this case, $\mu_G(i) = 1/i$. This means that the estimate of the policy values of policies from past iterations is not forgotten; despite each iteration having a different policy, the policies are similar to each other, thus an initialized estimate turns out to be

Figure 6.3 – Value of abstract policy built with AbsProb-RL. With $\mu_G(i) = 1$, three different estimation window functions $H(i)$ are compared.

better than an initialization with zeros. A third form, the *fixed* rate $\mu_G(i) = 0.05$, was also tested and performed as well as the global average.

Figure 6.4 illustrates the results to this second experiment, again showing the average policy value of 10 executions and $H(i) = 20i$. It is notable that the global average and the fixed rate performed significantly better than the average per iteration and eventually achieved values very close to the optimal.

Since $\mu_G(i) = 1/i$ presented the best result, the third experiment fixes the learning rate one more time for a revaluation of variations in size of the estimation window function H . Besides the incremental values from the first experiment, as the global average carries some previous estimates, fixed values for H had also been tested: the policy is improved every 20 or 50 episodes. The mean values of 20 executions to the same task are shown in Figure 6.5.

We notice that the curves with growing H increase monotonically. This is due to the fact that on every policy improvement, more episodes are used to estimate the gradient and therefore it is more accurate, increasing the odds that the policy improvement step has the right direction (as it is guaranteed in the model-based algorithm).

With fixed values of H , the gradient estimate is always done with the same number of episodes, not increasing its precision over time. For this reason, the curves with fixed H have an erratic behaviour after a while. Nevertheless, as H does not increase, much more improvement steps are made and in general we see that curve $H(i) = 50$ have higher

Figure 6.4 – Value of abstract policy built with AbsProb-RL. With $H(i) = 20i$, three different learning rates $\mu_G(i)$ are compared.

initial slope than the incremental ones. It is also worth noticing that these curves stop increasing value after reaching some threshold.

These experiments show that, given enough episodes, AbsProb-RL succeeds in finding an optimal abstract policy for the task. Learning rate as a global average arise as the best option and regarding the estimation window sizes, there is a trade-off between speed and convergence confidence.

Figure 6.5 – Value of abstract policy built with AbsProb-RL. With $\mu_G(i) = 1/i$, four different estimation window functions $H(i)$ are compared.

6.4 S2L-RL with AbsSarsa(λ)

Experiments in this section are on domain D_1 , which has 184 ground states and, depending on the goal position, tasks can be set with the number of abstract states ranging from 23 to 32. Figure 6.1 shows the map used, where a goal state s_g can be chosen among any state in a room. For example, this figure shows the task when the goal state is set $s_g = 4$; the abstract state $\sigma_1 = \text{inRoom}(X) \wedge \text{seeDoor}(Y) \wedge \text{appGoal}(Y) \wedge \text{nearGoal}$ includes the set $\mathcal{S}_{\sigma_1} = \{7\}$, whereas the abstract state $\sigma_2 = \text{inRoom}(X) \wedge \text{seeDoor}(Y) \wedge \text{appGoal}(Y)$ includes the set $\mathcal{S}_{\sigma_2} = \{21, 27, 37, 77, 101, 105, 137, 143, 165, 166, 173, 177, 184\}$. Depending on the goal position, the set of predicates describes differently each enumerated state, giving rise to different abstractions. We ran experiments with the following goal positions: 4, 12, 81, 95 and 181.

To assess how efficient the proposed RL framework is, we evaluated the performance of three types of RL agents. The first is an agent using the classical Sarsa(λ) algorithm (Algorithm 6) to solve a task, used as a baseline reference. The second uses the S2L-RL framework (Algorithm 16), but without any previous knowledge, whereas the third also uses S2L-RL to solve a target task, but with the aid of some abstract policies of previously solved source tasks. This experiment uses AbsSarsa(λ) as the algorithm at the abstract level (Algorithm 10) and Sarsa(λ) at the ground level of S2L-RL.

We consider a set of 5 tasks for the experiments, one for each goal state mentioned before. The parameters used for all agents are: $\epsilon = 0.2$, $\mu = 0.05$, $\gamma = 0.95$, $\lambda = 0.9$, $\psi_0 = 0$, $\nu = 0.95$ and $\Delta\tau = 0.05$. All these are constants and do not change during the learning process. The initial value of τ is set to $\tau_0 = 0$ for the agent without reuse and to $\tau_0 = 10$ for the agent with reuse of previous knowledge. This is because the former agent begins with two policies that are not ready and both are to be built, π_{gnd} and π_{abs} , therefore a initial value $\tau = 0$ assigns equal probabilities to both of them. On the other hand, the agent with reuse already starts with a number of past policies, which are already built. The higher initial value of τ ensures that these particular policies can be chosen more often in the beginning of the learning process.

For each task and agent, we run a total of 1000 episodes with a maximum number of steps per episode of 500. An episode starts in an initial state chosen following b^0 and it ends when the agent reaches the goal state or takes the maximum number of steps. Then a new episode is started and the process continues. When solving a task, the agent with reuse takes four abstract policies built previously with S2L-RL as its past policies. These policies are the solutions to all tasks, except the current one, from our set of 5 tasks. This process is repeated 30 times (totalizing 150 executions per agent) and the average accumulated reward, averaging all tasks, is shown in Figure 6.6. Here we only show the results for the target task, the learning with the source tasks is omitted for the agent with reuse.

Figure 6.6 – Comparison between Sarsa and S2L-RL with AbsSarsa(λ). S2L-RL simultaneously learns an abstract and a ground policy and can also reuse past abstract policies to accelerate learning.

We notice a significant improvement on the performance, especially on the initial portion of the learning process, when comparing S2L-RL to Sarsa. This is mainly due to the fact that the abstract policy is built much faster, as Section 4.1 shows in Figure 4.1 (which uses the same 5 goal states as this experiment). The abstract policy reaches a maximum value that is lower than the maximum of the optimal policy, but as it is reached earlier, it can provide a good guidance to the agent. This thus yields higher reward gains while the ground policy is not ready. This means that not only does the agent learn faster but also it accumulates in average more reward during the whole learning process.

Furthermore, when the agent takes advantage of previous knowledge, the performance is even better. As it is expected that the chosen abstraction hold some properties across different tasks, the past policies can guide the agent in the very beginning of the learning process, when almost no knowledge is acquired yet.

However it is important not to overuse these policies so the performance would be poorer in the long run. The measure W , the average reinforcement received per episode, gracefully controls the balance among all policies (the two being learned – ground and abstract – and the past ones), cutting off the policy reuse when it is no longer necessary. Along the 1000 episodes, the average use of the ground policy is around 86%, 10% for the past policies and the remaining 4% for the new abstract policy. The evolution of the usage of each policy over the time is shown in Figure 6.7. We can see that the contribution of past and abstract policies is concentrated in the first 200 episodes and thereafter the ground policy assumes the actions.

Figure 6.7 – Evolution of policy usage in S2L-RL, showing the average usage of each policy for 150 executions of S2L-RL.

6.5 S2L-RL with AbsProb-RL

In this section we conducted experiments using S2L-RL with AbsProb-RL at the abstract level. Additionally, this series of experiments compares the two different approaches to transfer learning with knowledge from multiple source tasks – case-based and knowledge generalization – described in Section 4.3. Actually, the focus here is mostly on the *transfer* of past policies. AbsProb-RL has a slower convergence than AbsSarsa(λ) or even the ground level depending on the size of the task, as we can notice comparing the two previous sections (plots on Section 6.3 go up to episode 10000 while plots on Section 6.4 go up to episode 1000). Therefore, it presents little help in speeding-up the ground learning in the current task, but it is useful for building policies to speed-up future tasks. Moreover, note that as use of π_{abs} tends to be little (Figure 6.7 shows it is used only 4% of the time), just the use of past policies provide a good speed-up in the learning process. The following experiments focus on this transfer feature.

Figure 6.8 – *Navigation Domain $\mathcal{D}_3$* – Black areas represent walls; darker cells, rooms; lighter cells, corridors and doors are marked with ‘d’. Each cell represents one state, except for wall and door cells. The goals of each source task are marked with ‘S’ and the goals of each target task with ‘T’.

The domain $\mathcal{D}_3$, depicted in Figure 6.8, has been chosen for these experiments because it is exactly the same (with the addition of a relational description) used in the paper which introduced the concept of a policy library and the PRQ-Learning algorithm (FERNÁNDEZ; VELOSO, 2006). We use this domain to perform some comparisons between S2L-RL and the results of this paper. It contains a total of 286 ground states and 45 distinct abstract states. Set C contains 15 doors, 177 rooms markers and 79 corridor markers.

Each experiment has two phases: *Knowledge Building* and *Target Learning*. In the first phase, the agent learns one or more abstract policies from a set of n *source tasks* $\Omega_1, \Omega_2, \dots, \Omega_n$. Two different approaches for encoding the policies are compared:

1. Generalized Policy: all source tasks are presented to the agent simultaneously and AbsProb-RL (Algorithm 14) is used to find one single generalized policy.
2. Policy Library: the source tasks are presented sequentially, one at a time; an abstract policy is found for each one with AbsProb-RL and it may or may not be included in the library depending on δ_W , according to PRPL (Algorithm 13).

Furthermore, as the δ_W parameter of the library builder algorithm influences the size of the library, four values are used: $\delta_W = 0$, $\delta_W = 0.25$, $\delta_W = 0.50$ and $\delta_W = 1.0$. We call the libraries formed with each value $L0$, $L25$, $L50$ and $L100$, respectively.

In the second phase, the agent faces a new task, called *target task*, which is different from the source tasks. S2L-RL is applied to this new task, i.e., the agent learns the optimal policy with Q -Learning but improves the performance of the learning process by reusing the policies learned in the first phase. Q -Learning is used at the ground level and in order to isolate only the transfer effects, we do not perform learning at the abstract level. The S2L-RL architecture uses a set of past policies. In the case of the generalized policy, the set contains just one element. In the case of the library, it contains the whole library. The S2L-RL without the learning at the abstract level is roughly the same as the PRQ-Learning algorithm, which in conjunction with the policy library built with PRPL, are the results of Fernández and Veloso (2006). Therefore, we compare here the performance of S2L-RL using a generalized policy with PRQ-Learning using PRPL.

Always starting in a random initial state (b^0 is a uniform probability distribution to all states), the agent tries to reach the goal state in an episode with a maximum of 100 steps. When it reaches the goal or fails to find it after 100 steps, the episode ends. The agent runs 2000 episodes ($K = 2000$) for each of the four library versions ($L0$, $L25$, $L50$, $L100$), for the generalized policy and also for the agent without reuse. Parameters used for S2L-RL are: $H = 100$, $\mu = 0.05$, $\gamma = 0.95$, $\epsilon = 0.05$, $\psi_0 = 1$, $v = 0.95$, $\tau_0 = 0$ and $\Delta\tau = 0.005$.

In our set of experiments, we also want to evaluate the evolution of the libraries size and the impact of the number of source tasks in the transfer. In order to do so, we first choose a total of 20 source tasks (whose goals are depicted as ‘S’ in Figure 6.8). Then, given a random sequence of these 20 tasks, we run experiments by gradually increasing n , from 1 to 20. In other words, the first experiment is run with $n = 1$ and the agent builds a generalized policy of just one task and also builds a library with just one policy. In the second experiment, $n = 2$, the next task is added and the agent builds a new generalized policy aggregating both tasks as well as a new library with size ≤ 2 , containing the previous policy and possibly the new policy that solves the second task (it can be added or not depending on the value of δ). This process continues until $n = 20$. Additionally, as the order the source tasks are presented may change the contents of a library, the whole process is repeated 4 times, always randomizing the ordering of the 20 source tasks.

Furthermore, in the *Target Learning* phase, 15 target tasks (whose goals are depicted as ‘T’ in Figure 6.8) are used and the results we show are the average of the results of these 15 tasks.

Figure 6.9 – *Transfer results for 20 source tasks* – Performance of the transfer of a generalized policy using knowledge of 20 source tasks, compared with transfer of policy libraries ($L0$, $L25$, $L50$, and $L100$) and Q -Learning without past knowledge (no transfer). Each point represents the average value of 100 executions and the error bars indicate the 95% confidence intervals based on the Student’s t -distribution.

Figure 6.9 shows the results when $n = 20$, i.e., the whole set of 20 source tasks is considered in knowledge building. After 20 source tasks, the average sizes of the libraries are: $|L0| = 1$, $|L25| = 5$, $|L50| = 11$ and $|L100| = 20$. Their performances are compared in terms of the cumulative average of the rewards received in each episode. The performance of an agent learning from scratch with Q-Learning, without using any kind of previous knowledge is also shown for comparison (label *No transfer* in Figure 6.9). This agent, as all others, uses the ϵ -greedy exploration strategy with a fixed value $\epsilon = 0.05$, a learning rate $\mu = 0.05$ and a discount factor $\gamma = 0.95$.

First of all, we can see that knowledge transfer methods do present a better performance. The amount of reward received by the agent in the first episodes is much higher when it reuses previous knowledge. This is because it used the already-learned policies to guide its exploration and thus led it to higher rewards even at the beginning of the learning process. Without reuse, on the other hand, the agent takes more time to explore the environment and thus behaves more erroneously to try to solve the task. Therefore, this shows that the previous policies did encode relevant information to the new tasks.

If we compare the two methods that use past knowledge, a better performance of the generalized policy than that of the policy library can be noticed, considering the average of all tasks. This is especially due to the fact that the library contains some policies that are not helpful at all – they may even be disadvantageous to the agent. Then the PRQ-Learning algorithm may take some time to notice that some policies are unhelpful and to assign a low probability of choosing them, whereas the generalized policy mostly presents some aid to the learning process. This suggests that, if we seek to always obtain the maximum reward, regardless of the number of episodes executed, the generalized policy is more suitable.

L0 was expected to present the poorest performance because it is the one that preserves less previous knowledge: just the policy of the first source task. All the other libraries, L25, L50 and L100, had quite a similar performance. Indeed, Student's t -tests were conducted pointwise and they show that the hypotheses that the learning curves of L25, L50 and L100 have all the same mean just after the 10^{th} episode can not be rejected with a confidence level greater than 95%. On the other hand, the hypothesis that they have the same mean as the generalized policy is rejected after the 10^{th} episode with a confidence level greater than 99.9%.

Figure 6.9 shows the results when there is knowledge from the solution of 20 source tasks, but what if n is smaller? Focusing on just one target task, Figure 6.10 shows the cumulative average of the rewards after 1000 episodes with different values of n , for the generalized policy, L25, L50 and L100. Looking at the evolution of n , we notice that the generalized policy presents a better performance regardless of the number of solved

source tasks. For all approaches, as n increases, the average reward also increases until it reaches a maximum value. Note that L25 and L50 do not include all the policies in their libraries and eventually stop including any policy at all, thus showing constant reward in some ranges of n . Additionally, the L100 curve is noticed to present a decreasing performance for higher values of n , i.e., when the library has too many policies in it, its performance may be poorer, an indication that the online policy selection process does present a significant overhead to the learning process.

Figure 6.10 – *Evolution of size of source task set* – Performance after 1000 episodes for different numbers of source tasks. Each point represents the average value of 100 executions.

These experiments assessed the transfer among tasks in the same domain. To show the transfer effect in tasks in a different domain, the knowledge from the same 20 source tasks is taken into consideration to improve performance in the learning of new tasks in a broader and never-seen-before domain. These tasks (and the domain) are illustrated in Figure 6.11.

The arithmetic mean of the cumulative average of the rewards after 100 runs in each of the 10 tasks is shown in Figure 6.12. The results are quite similar to the ones presented before: the generalized policy has an overall better result than the libraries. Library with $\delta = 0.25$ presented a slightly better performance than the ones with $\delta = 0.5$ and $\delta = 1.0$ (which contains all the 20 policies), corroborating the conclusions mentioned before that

Figure 6.11 – *Navigation domain $\mathcal{D}_4$* – 10 target tasks to evaluate transfer among tasks of different domains.

too many policies may affect the transfer performance. This experiment shows the power of a relational representation and abstract policies. They enabled the transfer process to a different domain (according to our definitions in Section 3.1), because the knowledge is kept at an abstract level.

Figure 6.12 – *Results of transfer to new domain* – Performance of policy reuse methods applying knowledge of 20 source tasks in a new domain. Each point represents the average value of 100 executions and the error bars indicate the 95% confidence intervals based on the Student's t -distribution.

7 CONCLUSION AND FUTURE WORK

In this work we tackled the problem of convergence speed of reinforcement learning agents. We saw that the use of state and action abstractions combined with policy transfer is effective for transfer learning. Abstraction is achieved by using a relational representation for the states and actions, using predicates carved by a human. Main advantage of using abstraction is that it provides generalization, an important feature for transfer learning; thereby, we are able to transfer policies between related tasks.

We contributed with 2 algorithms for building abstract policies: *AbsSarsa*(λ) and *AbsProb-RL*. *AbsSarsa*(λ) (Section 4.1) builds a deterministic abstract policy using value functions. It uses a slightly modified version of *Sarsa*(λ) (thus estimates action-value function Q) to build the abstract policy and although it may not find the optimal abstract policy, it succeeds in speeding-up learning, especially by reducing the usual initial low performance. We showed the abstract policy is learned fast enough so it can be used even in the current task being solved.

AbsProb-RL (Section 4.2.3) takes a step further and uses direct policy search to find the optimal stochastic abstract policy. It is based on planning algorithm Policy Iteration, using Monte Carlo techniques for online estimation of policy values and calculating gradients to improve the policy. The stochastic policies it builds grant more flexibility to the transferred knowledge and help preventing negative transfer. Moreover, when dealing with multiple source tasks, *AbsProb-RL* also has the ability to build a *generalized policy*, which encodes knowledge that tries to satisfy all source tasks, then it is more likely to be useful to a new target task.

We have also contributed with an agent architecture, namely S2L-RL: Simultaneous Two-layer Reinforcement Learning (Chapter 5). It has two levels of hierarchy: an abstract and a ground level. In the abstract level the agent maintains past policies and also learns a new abstract policy, using *AbsSarsa*(λ) or *AbsProb-RL*. These policies (past and new abstract) can be used to bias the exploration the agent needs to conduct, therefore they influence on the building of the optimal policy at the ground level. The idea of transferring an abstract policy is interesting because the abstract learner does not need to know the entire state and action spaces of the target task; it is only necessary that the ground learner knows how to translate abstract actions into ground actions and ground states into abstract states.

The combination of the architecture with algorithms to build abstract policies is our proposed solution to the transfer learning problem stated in Section 3.1. The following characteristics are their main differences from existing transfer learning methods:

- Make use of state and action abstraction;
- Abstract policy is the transferred knowledge (deterministic for AbsSarsa(λ) and stochastic for AbsProb-RL);
- Enable transfer between tasks with different state and action spaces, provided that they are from the same domain class (refer to Section 3.1);
- Allows multiple source tasks.

Our experiments showed that S2L-RL do improve performance of a RL agent in a robotic navigation domain. This improvement is most noticeable by a higher slope in the agent's performance, i.e., the performance at the early stages of the learning process is improved. The abstract policies obtained from source tasks that are used to accelerate learning on a target task effectively guide the agent when knowledge about the current task is minimal.

We also performed comparisons with another approach to multiple source tasks, the use of a policy library, in contrast to a single generalized policy. We conclude that in general the generalized policy presents better transfer ratio (ratio of the total reward accumulated by the transfer learner and the total reward accumulated by the non-transfer learner), mainly due to the management effort needed when using a large library.

This dissertation concludes the author's master's course. During the course, other works were published:

- “Avaliação de Políticas Abstratas na Transferência de Conhecimento em Navegação Robótica” (BEIRIGO et al., 2012) presents Qab-Learning algorithm;
- “Online Learning of Abstract Stochastic Policies with Monte Carlo” (KOGA; SILVA; COSTA, 2013a) presents the AbsProb-RL algorithm;
- “Speeding-up reinforcement learning through abstraction and transfer learning” (KOGA et al., 2013) presents AbsSarsa(λ) algorithm and part of the S2L-RL architecture;
- “Reusing Risk-Aware Stochastic Abstract Policies in Robotic Navigation Learning” (SILVA et al., 2013) presents how to incorporate risk-awareness into abstract policies;

- “Using Stochastic Abstract Policies to Generalize Knowledge and Improve Reinforcement Learning” (KOGA; SILVA; COSTA, 2013b) presents part of the S2L-RL architecture and analyzes the ability of AbsProb-PI to create generalized policies (this paper actually was just submitted for publication).

Extensions of this dissertation that can be addressed by future research are described here:

- **Automatic features discovery.** The predicates used to describe states and actions in this work are hand-crafted and also the abstraction level is fixed. Seeking to solve this problem, researchers have investigated how to define an abstraction that guarantee a good policy (PAINTER-WAKEFIELD; PARR, 2012; GERAMIFARD et al., 2011; SUN et al., 2011) and how to automatically, or semi-automatically, obtain features (DAS; DILL, 2002; JONG; STONE, 2005).
- **Continuous variables.** Future work also involves dealing with problems with continuous state variables, which requires the use of functions approximators, such as CMAC (cerebellar model arithmetic computer). This enables experiments on new domains such as the RoboCup simulation soccer (STONE; SUTTON; KUHLMANN, 2005; STONE et al., 2006) and real-world problems of robotic navigation (BACCA; SALVI; CUFÍ, 2011; WU et al., 2013).
- **Other strategies for generalizing knowledge from multiple source tasks.** AbsProb-RL generalizes knowledge by finding a policy with higher value in average for all tasks. We can investigate other forms for a better generalization, considering risk of solving each task for example, instead of being risk-neutral for all tasks. We have already published a work with discussions on the risk-sensitivity of policies (SILVA et al., 2013).
- **Lifelong learning.** Although AbsProb-RL allows extracting knowledge from multiple source tasks, it requires all source tasks to be available at the same time. Future work can investigate the combination with Lifelong learning methods, where an agent faces a sequence of tasks. In this scenario, it is desired that the agent accumulates knowledge and learns to adapt to new situations that arise. Transfer methods are a key component and there are some works in this field in the RL setting (SUTTON; KOOP; SILVER, 2007).

REFERENCES

- ABERDEEN, D. A (revised) survey of approximate methods for solving partially observable markov decision processes. *National ICT Australia, Canberra, Australia, Tech. Rep*, 2003.
- ASADA, M.; NODA, S.; TAWARATSUMIDA, S.; HOSODA, K. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, Kluwer Academic Publishers, v. 23, n. 2-3, p. 279–303, 1996.
- ASADI, M.; HUBER, M. Effective control knowledge transfer through learning skill and representation hierarchies. In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI '07)*. 2007. v. 7, p. 2054–2059.
- BACCA, B.; SALVI, J.; CUFÍ, X. Appearance-based mapping and localization for mobile robots using a feature stability histogram. *Robotics and Autonomous Systems*, v. 59, n. 10, p. 840–857, 2011.
- BEIRIGO, R. L.; PEREIRA, F. A.; KOGA, M. L.; MATOS, T.; SILVA, V. F. da; COSTA, A. H. R. Avaliação de políticas abstratas na transferência de conhecimento em navegação robótica. *Revista de Sistemas e Computação*, v. 2, p. 17–25, 2012.
- BELLMAN, R. E. A markovian decision process. 1957.
- BIANCHI, R. A.; ROS, R.; MANTARAS, R. L. D. Improving reinforcement learning by using case based heuristics. In: *Case-Based Reasoning Research and Development - 8th International Conference on Case-Based Reasoning, ICCBR 2009 Seattle, WA, USA, July 20-23, 2009 Proceedings*. New York: Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5650). p. 75–89.
- BLOCKEEL, H.; DE RAEDT, L. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, v. 101, n. 12, p. 285–297, 1998.
- BOUTILIER, C.; DEARDEN, R.; GOLDSZMIDT, M. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, Elsevier, v. 121, n. 1, p. 49–107, 2000.
- CAO, X.-R.; FANG, H.-T. Gradient-based policy iteration: an example. In: *Proceedings of the 41st IEEE Conference on Decision and Control*. 2002. v. 3, p. 3367–3371.
- CASTRO, D. D.; TAMAR, A.; MANNOR, S. Policy gradients with variance related risk criteria. In: *Proceedings of the 29th International Conference on Machine Learning (ICML '12)*. 2012. p. 935–942.
- CELIBERTO JR., L. A.; MATSUURA, J. P.; DE MANTARAS, R. L.; BIANCHI, R. A. C. Using cases as heuristics in reinforcement learning: a transfer learning application. In: *Proceedings of the 22th International Joint Conference on Artificial Intelligence (IJCAI '11)*. 2011. p. 1211–1217.

- COLOMBETTI, M.; DORIGO, M. *Robot shaping: Developing situated agents through learning*. International Computer Science Institute (ICSI), Berkeley, CA, 1993. Technical Report TR-92-040.
- CONN, K.; PETERS, R. Reinforcement learning with a supervisor for a mobile robot in a real-world environment. In: *Proceedings of the 7th International Symposium on Computational Intelligence in Robotics and Automation (CIRA '07)*. 2007. p. 73–78.
- CROONENBORGH, T.; DRIESSENS, K.; BRUYNOOGHE, M. Learning relational options for inductive transfer in relational reinforcement learning. In: *Proceedings of the 17th Conference on Inductive Logic Programming (ILP '07)*. 2007. p. 88–97.
- DAS, S.; DILL, D. L. Counter-example based predicate discovery in predicate abstraction. In: SPRINGER. *Formal Methods in Computer-Aided Design*. 2002. p. 19–32.
- DEISENROTH, M.; RASMUSSEN, C. Pilco: A model-based and data-efficient approach to policy search. In: *Proceedings of the 28th International Conference on Machine Learning (ICML '11)*. 2011. p. 465–472.
- DEVLIN, S.; KUDENKO, D. Dynamic potential-based reward shaping. In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '12)*. 2012. p. 433–440.
- DIETTERICH, T. G. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, v. 13, p. 227–303, 2000.
- DZEROSKI, S.; DE RAEDT, L.; DRIESSENS, K. Relational reinforcement learning. *Machine Learning*, v. 43, n. 1/2, p. 7–52, 2001.
- FERNÁNDEZ, F.; GARCÍA, J.; VELOSO, M. Probabilistic policy reuse for inter-task transfer learning. *Robotics and Autonomous Systems*, Elsevier B.V., v. 58, n. 7, p. 866–871, 2010.
- FERNÁNDEZ, F.; VELOSO, M. Probabilistic policy reuse in a reinforcement learning agent. In: *Proceedings of the fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*. 2006. p. 720–727.
- GERAMIFARD, A.; DOSHI, F.; REDDING, J.; ROY, N.; HOW, J. Online discovery of feature dependencies. In: *Proceedings of 28th International Conference on Machine Learning (ICML '11)*. 2011. p. 881–888.
- HAUSKRECHT, M.; MEULEAU, N.; KAEHLING, L. P.; DEAN, T.; BOUTILIER, C. Hierarchical solution of markov decision processes using macro-actions. In: *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI '98)*. 1998. p. 220–229.
- HOWARD, R. *Dymanic Programming and Markov Processes*. Cambridge, MA: MIT Press, 1960.

- JAAKKOLA, T.; SINGH, S. P.; JORDAN, M. I. Reinforcement learning algorithm for partially observable markov decision problems. *Advances in Neural Information Processing Systems*, MIT Press, v. 7, p. 345, 1994.
- JONG, N. K.; STONE, P. State abstraction discovery from irrelevant state variables. In: *Proceedings of the 19th International Joint Conference on Artificial intelligence (IJCAI '05)*. 2005. p. 752–757.
- KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, v. 4, p. 237–285, 1996.
- KALOS, M. H.; WHITLOCK, P. A. *Monte carlo methods*. 2nd. ed. Weinheim: WILEY-VCH, 2008.
- KERSTING, K.; VAN OTTERLO, M.; DE RAEDT, L. Bellman goes relational. In: *Proceedings of the 21st International Conference on Machine learning (ICML '04)*. 2004. p. 59.
- KOGA, M. L.; SILVA, V. F. da; COSTA, A. H. R. Online learning of abstract stochastic policies with monte carlo. In: *Proceedings of the 7th Workshop de Tecnologia Adaptativa (WTA '13)*. 2013.
- KOGA, M. L.; SILVA, V. F. da; COSTA, A. H. R. Using stochastic abstract policies to generalize knowledge and improve reinforcement learning. Submitted to IEEE Transactions on Cybernetics. 2013.
- KOGA, M. L.; SILVA, V. F. da; COZMAN, F. G.; COSTA, A. H. R. Speeding-up reinforcement learning through abstraction and transfer learning. In: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '13)*. 2013. p. 119–126.
- KONIDARIS, G.; SCHEIDWASSER, I.; BARTO, A. Transfer in reinforcement learning via shared features. *Journal of Machine Learning Research*, Microtome Publishing, v. 13, p. 1333–1371, 2012.
- LI, L.; WALSH, T. J.; LITTMAN, M. L. Towards a unified theory of state abstraction for MDPs. In: *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics (ISAIM '06)*. 2006. p. 531–539.
- LI, Y.; YIN, B.; XI, H. Finding optimal memoryless policies of POMDPs under the expected average reward criterion. *European Journal of Operational Research*, Elsevier, v. 211, n. 3, p. 556–567, 2011.
- LITTMAN, M. L. Memoryless policies: theoretical limitations and practical results. In: *Proceedings of the 3rd International Conference on Simulation of Adaptive Behavior: from animals to animats 3*. 1994. p. 238–245.
- LOCH, J.; SINGH, S. P. Using eligibility traces to find the best memoryless policy in partially observable markov decision processes. In: *Proceedings of the 13th International Conference on Machine Learning (ICML '98)*. 1998. p. 323–331.

- MARTÍN, M.; GEFNER, H. Learning generalized policies from planning examples using concept languages. *Applied Intelligence*, p. 9–19, 2004.
- MATOS, T.; BERGAMO, Y. P.; SILVA, V. F. da; COZMAN, F. G.; COSTA, A. H. R. Simultaneous abstract and concrete reinforcement learning. In: *Proceedings of the 9th Symposium on Abstraction, Reformulation and Approximation (SARA '11)*. 2011. p. 82–89.
- MATOS, T.; BERGAMO, Y. P.; SILVA, V. F. da; COSTA, A. H. R. Stochastic abstract policies for knowledge transfer in robotic navigation tasks. In: *Advances in Artificial Intelligence*. [S.l.]: Springer, 2011. p. 454–465.
- MEHTA, N.; NATARAJAN, S.; TADEPALLI, P.; FERN, A. Transfer in variable-reward hierarchical reinforcement learning. *Machine Learning*, Springer, v. 73, n. 3, p. 289–312, 2008.
- METROPOLIS, N.; ULAM, S. The monte carlo method. *Journal of the American statistical association*, Taylor & Francis Group, v. 44, n. 247, p. 335–341, 1949.
- MORALES, E. F. Scaling up reinforcement learning with a relational representation. In: *Proceedings of the Workshop on Adaptability in Multi-agent Systems*. 2003. p. 15–26.
- NAVARRO-GUERRERO, N.; WEBER, C.; SCHROETER, P.; WERMTER, S. Real-world reinforcement learning for autonomous humanoid robot docking. *Robotics and Autonomous Systems*, Elsevier B.V., v. 60, n. 11, p. 1400–1407, 2012.
- NG, A. Y. *Shaping and policy search in reinforcement learning*. Thesis (Ph.D.) — University of California, 2003.
- NG, A. Y.; JORDAN, M. Pegasus: a policy search method for large MDPs and POMDPs. In: *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*. 2000. (UAI '00), p. 406–415.
- PAINTER-WAKEFIELD, C.; PARR, R. Greedy algorithms for sparse reinforcement learning. In: *Proceedings of the 29th International Conference on Machine Learning (ICML '12)*. 2012. p. 1391–1398.
- PAN, S. J.; YANG, Q. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, v. 22, n. 10, p. 1345–1359, oct. 2010.
- PAZIS, J.; PARR, R. Generalized value functions for large action sets. In: *Proceedings of the 28th International Conference on Machine Learning (ICML '11)*. 2011. p. 1185–1192.
- PUTERMAN, M. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. [S.l.]: John Wiley & Sons, Inc., 1994.
- QUINLAN, J. R. *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.
- RICHARDSON, M.; DOMINGOS, P. Markov logic networks. *Machine learning*, Springer, v. 62, n. 1-2, p. 107–136, 2006.

- RUMMERY, G. A.; NIRANJAN, M. *On-line Q-learning using connectionist systems*. Engineering Department, University of Cambridge, 1994. Technical Report CUED/F-INFENG/TR 166.
- RUSSELL, S.; NORVIG, P. *Artificial intelligence: A Modern Approach*. 2nd. ed. New Jersey: Prentice Hall, 2003.
- SELFDRIDGE, O. G.; SUTTON, R. S.; BARTO, A. G. Training and tracking in robotics. In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI '85)*. 1985. p. 670–672.
- SILVA, V. F. da; KOGA, M. L.; COZMAN, F. G.; COSTA, A. H. R. Reusing risk-aware stochastic abstract policies in robotic navigation learning. In: *Proceedings of the 17th annual RoboCup International Symposium*. 2013.
- SILVA, V. F. da; PEREIRA, F. A.; COSTA, A. H. R. Finding memoryless probabilistic relational policies for inter-task reuse. In: *Proceedings of the 14th International Conference on Information Processing and Management of Uncertainty (IPMU '12)*. 2012. (Communications in Computer and Information Science, v. 298), p. 107–116.
- SILVER, D.; VENESS, J. Monte-carlo planning in large POMDPs. In: LAFFERTY, J.; WILLIAMS, C. K. I.; SHAW-TAYLOR, J.; ZEMEL, R.; CULOTTA, A. (Ed.). *Advances in Neural Information Processing Systems 23*. [S.l.: s.n.], 2010. p. 2164–2172.
- SINGH, S. P.; JAAKKOLA, T.; JORDAN, M. I. Learning without state-estimation in partially observable markovian decision processes. In: *Proceedings of the 11th International Conference on Machine Learning (ICML '94)*. 1994. v. 31, p. 37.
- STONE, P.; KUHLMANN, G.; TAYLOR, M. E.; LIU, Y. Keepaway soccer: From machine learning testbed to benchmark. In: *RoboCup 2005: Robot Soccer World Cup IX*. Berlin: Springer Verlag, 2006. p. 93–105.
- STONE, P.; SUTTON, R. S.; KUHLMANN, G. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, Sage Publications, v. 13, n. 3, p. 165–188, 2005.
- STULP, F.; SIGAUD, O. Path integral policy improvement with covariance matrix adaptation. In: *Proceedings of the 29th International Conference on Machine Learning (ICML '12)*. 2012. p. 281–288.
- SUN, Y.; GOMEZ, F.; RING, M.; SCHMIDHUBER, J. Incremental basis construction from temporal difference error. In: *Proceedings of the 28th International Conference on Machine Learning (ICML '11)*. 2011. p. 481–488.
- SUTTON, R. S. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In: *Proceedings of the 7th International Conference on Machine Learning (ICML '90)*. 1990. p. 216–224.
- SUTTON, R. S. Planning by incremental dynamic programming. In: *Proceedings of the 8th International Workshop on Machine Learning*. 1991. p. 353–357.

- SUTTON, R. S. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In: TOURETZKY, D. S.; MOZER, M. C.; HASSELMO, M. E. (Ed.). *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*. 1996. p. 1038–1044.
- SUTTON, R. S.; BARTO, A. G. *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press, 1998.
- SUTTON, R. S.; KOOP, A.; SILVER, D. On the role of tracking in stationary environments. In: *Proceedings of the 24th International Conference on Machine Learning (ICML '07)*. 2007. p. 871–878.
- SUTTON, R. S.; PRECUP, D.; SINGH, S. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, Elsevier Science Publishers Ltd., Essex, UK, v. 112, n. 1-2, p. 181–211, 1999.
- TAYLOR, M. E.; STONE, P. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, JMLR.org, v. 10, p. 1633–1685, 2009.
- TAYLOR, M. E.; STONE, P.; LIU, Y. Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, v. 8, n. 1, p. 2125–2167, 2007.
- TAYLOR, M. E.; WHITESON, S.; STONE, P. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In: *Proceedings of the 8th annual conference on Genetic and evolutionary computation (GECCO '06)*. 2006. p. 1321–1328.
- TAYLOR, M. E.; WHITESON, S.; STONE, P. Transfer via inter-task mappings in policy search reinforcement learning. In: *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '07)*. 2007. p. 37:1–37:8.
- THORNDIKE, E.; WOODWORTH, R. S. The influence of improvement in one mental function upon the efficiency of other functions.(i). *Psychological review*, The Macmillan Company, v. 8, n. 3, p. 247, 1901.
- THRUN, S.; MITCHELL, T. M. Lifelong robot learning. *Robotics and Autonomous Systems*, v. 15, n. 1–2, p. 25–46, 1995.
- TORREY, L. *Relational transfer in reinforcement learning*. Thesis (Ph.D.) — University of Wisconsin-Madison, 2009.
- TORREY, L.; SHAVLIK, J. Transfer Learning. *Handbook of Research on Machine Learning Applications*, IGI Global, p. 1–22, 2009.
- VAN OTTERLO, M. Reinforcement learning for relational MDPs. In: *Proceedings of the 13th Machine Learning Conference of Belgium and the Netherlands*. 2004. p. 138–145.
- VAN OTTERLO, M. *The logic of adaptive behavior: knowledge representation and algorithms for adaptive sequential decision making under uncertainty in first-order and relational domains*. Amsterdam: IOS Press, 2009.

WATKINS, C. J. C. H. *Learning from delayed rewards*. Thesis (Ph.D.) — University of Cambridge, 1989.

WILLIAMS, J. K.; SINGH, S. Experimental results on learning stochastic memoryless policies for partially observable markov decision processes. In: MIT PRESS. *Proceedings of the 1998 conference on Advances in neural information processing systems II*. 1998. p. 1073–1079.

WU, H.; TIAN, G. hui; LI, Y.; ZHOU, F. yu; DUAN, P. Spatial semantic hybrid map building and application of mobile service robot. *Robotics and Autonomous Systems*, n. 0, p. —, 2013. In press.