SAPIENZA
UNIVERSITÀ DI ROMA

Reinforcement Learning in Plan Space

Matteo Leonetti

a thesis submitted for the degree of
Doctor of Research (Ph.D)
in Computer Engineering

December 2010

Contents

Contents ii

List of Figures v

Aknowledgements vii

Introduction ix

Aim . . . xi

Contributions . . . xi

Summary . . . xiv

I Background and Related Work 1

1 Reinforcement Learning 3

1.1 Markov Decision Processes . . . 4

1.1.1 Policies and rewards . . . 5

1.1.2 Value functions . . . 6

1.1.3 The Bellman operator . . . 7

1.2 Value prediction . . . 8

1.2.1 Every-visit Monte Carlo . . . 8

1.2.2 Temporal Difference methods . . . 9

1.3 Semi-Markov Decision Processes . . . 11

1.4 Hierarchical Reinforcement Learning . . . 11

1.4.1 Introduction to Temporal Abstractions . . . 12

1.4.2 Hierarchy of Abstract Machines . . . 12

1.4.3 ALisp . . . 13

1.5 Non-Markovian Decision Processes . . . 14

1.5.1 Policies for POMDPs . . . 15

1.5.2 Value prediction in NMDPs 16
1.5.3 Learning equilibria 17
1.5.4 A theoretically sound algorithm for local searching 19

2 Plan Representation and Petri Nets **21**
2.1 Overview of FSA and PNs for plan representation 21
2.2 Petri Nets and related formalisms 24
2.2.1 Introduction to Petri Nets 24
2.3 Petri Net Plans 25
2.4 A model-based approach with Petri Nets 29
2.5 Summary of related work 30

II Combining Planning and Learning **33**

3 From Plans to Controllable Stochastic Processes **35**
3.1 Plan Representation 35
3.1.1 The main components 36
3.1.2 Plan schemas 38
3.1.3 On events and conditions 40
3.2 Learning Framework 41
3.2.1 Definition of a controllable stochastic process 42
3.2.2 Markovian and non-Markovian rewards 47
3.3 Summary 48

4 Learning Algorithms **51**
4.1 What is wrong with *direct* RL 51
4.1.1 Consistency 52
4.1.2 Action values 53
4.2 The algorithm: SoSMC 54
4.2.1 Exploration: gathering information 56
4.2.2 Assessment 58
4.3 Choosing the parameters 60
4.4 Credit assignment for parallel execution 61
4.5 Summary 62

5 LearnPNP **63**
5.1 Learning in Petri Nets 64

5.1.1 Petri Net Plans vs. State Charts 64
5.2 Introducing non-deterministic choice points 65
5.3 The operational semantics 66
5.4 The learning problem 67
5.4.1 Definition of the controllable process 67
5.5 Other possible extensions to PNP 68
5.6 Summary 74

III Experimental Results **75**

6 Grid Worlds **77**
6.1 Parr and Russell's Grid World 77
6.1.1 The Plan 78
6.1.2 Experimental results 79
6.2 About learning equilibria 81
6.3 SoSMC and Sarsa more closely 84
6.4 Sutton's Grid World 85

7 Keepaway **89**
7.1 Task Definition 90
7.2 Single-agent learning 91
7.3 Multi-agent learning 93

8 Conclusions **97**

Bibliography **99**

List of Figures

1 Modelling, planning, and learning. x

2.1 Nodes of a Petri Net 24
2.2 Ordinary and sensing actions in a Petri Net Plan 26
2.3 The `sequence` operator 26
2.4 The `interrupt` operator 27
2.5 The `fork` and `join` operators 28
2.6 A Petri Net Plan for a football playing robot 29

3.1 A simple behaviour for a football playing robot 36
3.2 A non-deterministic choice point in the domain of RoboCup Soccer . . 39
3.3 A simple example of a sub-procedure 40
3.4 The non deterministic choice point from the example of Figure 3.2 . . . 43
3.5 The transformation of the non deterministic choice point into an TNMDP 43
3.6 Example of a non Markovian reward 48
3.7 A different structure for the formed example that allows to distinguish the actual state 49

4.1 An example on consistent exploration 52
4.2 An NMDP in which the exploration can damage the current estimation of the value function 53
4.3 A simple example of an NMDP (a). The four policies return a reward normally distributed whose means and standard deviations are shown in (b). The evolution of the Q-function for the first state (actions A1 and B1) is represented in Figure (c), while for the second state (actions A2 and B2) is represented in Figure (d). 58

5.1 An example of a simple choice point between two ordinary actions . . 66
5.2 Example of a non Markovian reward 69

5.3 A tree structure to disambiguate choices 70
5.4 A memory place, to remember previous choices 70
5.5 Marking after WalkStraight has been executed 71
5.6 Marking after Turn has been executed 71
5.7 A compact way to count loops 72
5.8 A PNP for a domain with four states and four actions 73
5.9 A LearnPNP for more compactly represents the same behaviour 73

6.1 Parr and Russell's Grid World 78
6.2 The plan for Parr and Russell's Grid World 78
6.3 Average reward across 200 runs, on the short term for different controllers. SoSMC is evaluated without any exploration in the second phase. 79
6.4 Average reward across 200 runs on the long term allowing exploration in the second phase of SoSMC 80
6.5 Q-learning with ϵ-greedy exploration 82
6.6 Sarsa with ϵ-greedy exploration 82
6.7 Q-learning with SoftMax exploration, $\tau = 0.01$ 83
6.8 Sarsa with SoftMax exploration, $\tau = 0.01$ 83
6.9 Sarsa with SoftMax exploration, $\tau = 1$ 84
6.10 SoSMC with SoftMax exploration 85
6.11 Sarsa(λ) 86
6.12 Sutton's Grid World 86
6.13 Results for Sutton's domain 87

7.1 The field for Keepaway 90
7.2 The procedural part of the plan for playing Keepaway 91
7.3 Single agent learning the passing behaviour 92
7.4 Positions available to the agents in the learning task 93
7.5 Multi-agent Keepaway, learning positioning 94
7.6 The value of each action during a particular run. Note how the first and third agent switch roles between going to corner 3 and middle point 1 95

Aknowledgements

Thanks to my tutor, Dr Luca Iocchi, for his constant support throughout this long, as much as exciting, part of my life. He has been on my side since my first clumsy attempts at programming AIBOs for the RoboCup team SPQR, to appointing me the leader of that team, and for both my master's and PhD thesis. Along the same line I would like to thank Prof. Daniele Nardi, for the trust he has always placed in me, and his dedication to teaching and research.

One of the best opportunities this PhD has given me has been the chance to work with amazing people from all over the world. Among them, I would like to thank Prof. Andrew Barto, who kindly agreed in having me as a visiting scholar at the University of Massachusetts Amherst. The insights about RL from him, his co-director Prof. Sridhar Mahadevan, and the members of their group have proved fundamental to this work. Prof. Barto, by showing how being an outstanding and renown scientist can be matched by kindness and inter-personal skills, gives hope to us all. Among the ALL (Autonomous Learning Laboratory) members, I would like to thank in particular George Konidaris, who has taken care of me during my whole stay, both scientifically and providing a continuous source of fun.

During my last year, I have also had the chance to work at the University of Edinburgh with Dr Subramanian Ramamoorthy, whose helpfulness and long term vision I could greatly appreciate. Our weekly discussions have provided a huge boost to my research, and his different background has been of a crucial benefit in broadening my view of Machine Learning, and AI.

Thanks to those who helped me, supported me, taught me a lot, laughed with me, and kept me going. And to Flavia for all of the above.

Introduction

Robotic applications are characterised by highly dynamic domains, where the agent has neither full control of the environment, nor full observability. Decision making, in such domains, requires high reactivity and adaptation to uncertainty. Several sources of computational complexity of planning are simultaneously present in robotic tasks, such as: time constraints, parallel action execution, multiple agents, exogenous events, time extended actions, partial observability, and noise on sensors and actuators. We target our approach at such domains, considering the major difficulty that arises: the knowledge about the environment cannot be complete, and even the descriptions of the state space are quite arbitrary, made by the designers at their best to fit their needs. This has implications at every level, from modelling to execution, and we are aware of no other framework that does not make restrictions in any of the respects we mentioned.

As the problem grows complex, the solution must clearly be approximated. Optimality is not the focus of this work, we rather aim at improving the agent's behaviour as much as the current amount of knowledge allows, and at requiring as little knowledge about the domain as possible. This is motivated by the observation that, at the moment, robotic agents can rarely have a deep understanding of what surrounds them. The methods designed for optimal control give good results when the environment is the agent itself and little more: reading joints, controlling grasping, balancing a pole, is not the same as playing football, driving a car, or tending elderly people.

As a consequence of the difficulty of correctly modelling the environment, automatic planning can result in brittle plans that might need to be repaired or frequently recomputed. Execution Monitoring (Pettersson, 2005) is a field of research that deals with unreliable planning, but being able to recognise and react to failures might be not enough: an intelligent agent should learn from its mistakes and avoid them as much as possible in the future. The Reinforcement Learning (RL, Sutton and Barto, 1998) paradigm suits perfectly to this scenario, since it is

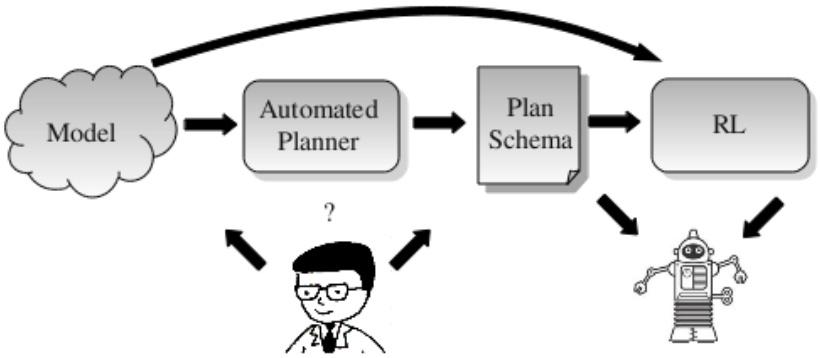


Figure 1: Modelling, planning, and learning.

based on trial-and-error and the agent can have little knowledge about the domain prior to execution. Learning alone, however, has proved to be impractical as the size of the domain scales. In order to address this issue, two methods are usually employed: generalising with function approximation, and simplifying the task by breaking it hierarchically, abstracting the least relevant details of the domain. We follow the latter approach, although at a different level than the work on state abstraction (Li et al, 2006; Giunchiglia and Walsh, 1992; Rogers et al, 1991) and Relational RL (Dzeroski et al, 2001), since we are going to intervene not on the model that generates the plan, but on the plan itself.

A simple representation of the possible strategies for the creation of agent behaviours is depicted in Figure 1. Traditional, *flat*, Reinforcement Learning relies on a state description to improve the agents' performance from experience. *Model-free* methods can learn without a complete model of the environment. The input to the learning algorithm, on the path from the designer to the RL block via the model, can be interpreted as either a problem formulation in terms of state description and reward, or a *model-based* approach. Another possible strategy is creating a complete model and feed it to a planner, whose output may or may not be subject to learning itself. Hierarchical RL (HRL, Barto and Mahadevan, 2003; Marthi et al, 2005; Dietterich, 2000) takes as input a partially specified behaviour, and can be placed on top of a planning process, although who produces such a behaviour is usually not specified. HRL has been developed on the model of MDPs as a way to restrict the set of *policies* considered. As a consequence, those frameworks only allow full observability and single agents, while we extend our work to partial knowledge and multi-agent systems.

Aim

The main objective of this work is devising and implementing a framework to improve agent behaviours in environments that challenge the assumptions of the other existing methodologies. We combine planning and reinforcement learning in a novel way that can be interpreted from three different perspectives:

- Automated planning: increase the robustness of plans in the case of partial knowledge, and optimise the behaviour on the aspects not completely modelled.
- Reinforcement learning: constrain the search space and speed up learning. Since the exploration considers only those actions that the planner has regarded as reasonable, time is not wasted on costly exploration of all the possible action sequences.
- Human designer: the framework allows the designer to act on the plan at an intelligible level (as opposed, for instance, to the weights of a neural network), and cooperate with the learning algorithm in a possibly cyclic process of iterative improvements.

Contributions

The main contributions of this work can be briefly summarised as follows:

1. Definition of a learning problem in a state space derived from a partial specification of the agent's behaviour
2. Since the controllable stochastic process generated by the framework is non-Markovian, we devise an algorithm of stochastic search for learning that does not rely on the Markov property.
3. The problem of representing plans is addressed together with the possibility opened up by learning. Petri Nets (PN) are used to overcome the limitations of finite state machines in terms of memory, compactness, and full parallelism of actions. Doing so, a novel way of learning in a language based on Petri Nets is defined.
4. We carry out an experimental evaluation on different domains, implementing a publicly available software library for learning with a language based

on Petri Nets, and shedding some light on non-Markovian domains. We compare our algorithm with the traditional ones that make use of value functions in tabular form.

The first contribution is the definition of a learning problem in plan space. It applies to a quite general category of plans that are represented as hierarchical state machines, with parallel action execution, sensing, loops, and interrupts. By allowing machine learning to help solve the planning problem, the system can adapt to unknown environments and the demand on the correctness of the model used for planning can be loosened. The learning algorithm does not rely on a model on its own, since its state space is derived directly from the plan. The balance between planning and learning is up to the designer and to the capability of the underlying planner: from few choice points to the full search in plan space. This also allows the system to benefit from the advances in modelling and planning, as much as to supply a powerful tool to the human designer for providing his knowledge about the task in the form of a sketch of a plan.

The vast majority of RL methods rely on Markov Decision Processes (Puterman, 1994) as a model of the domain. Assuming the Markov property means assuming the agent has complete knowledge of the aspects it would like to predict, and that affect how the reward is collected. As mentioned at the beginning, the domains we address do not allow for such an assumption to be fulfilled. Indeed, the controllable stochastic process generated from the plan is in general non-Markovian. An analogous problem is faced in Partially Observable MDPs (Monahan, 1982; Kaelbling et al, 1998), where it is assumed an underlying Markovian system, but not all of its states can be distinguished. The observations over the hidden MDP form a non-Markovian process, and it has already been noticed (Littman, 1994; Perkins, 2002; Crook, 2006) that memory-less policies on such a process can be quite appealing. Most of the techniques for POMDPs rely on a description of the state space and attempt some form of state estimation. *Belief states* are usually used to summarise the experience as a distribution over the possible states of the MDP. Again this requires a description of such an MDP, that is unfortunately not available or excessively complex. We follow the research line on *direct* RL over the observations of POMDPs, but we apply it to a different state space, since it is already the result of a planning process. This makes the system to control much more versatile than what could be achieved with memory-less policies on observations. The second contribution of this dissertation is a stochastic search algorithm for learning in non-Markovian domains. Non-Markovian

processes are the most general, therefore as a consequence of the *no free lunch theorem for optimisation* (Wolpert and Macready, 1997) no search algorithm in policy space can perform better that random searching on all the possible processes. In our algorithm, Stochastic Search Monte Carlo (SoSMC), we rely on the fact that the subset of all possible non-Markovian processes generated by our method are subject to the constraints and the regularities that real-world domains impose. Although there is no theoretical guarantee, as usual in stochastic search with noisy evaluation functions, that the algorithm performs better than the traditional ones (only Perkins's MCESP by is sound on NMDPs though, and it is a local method) we provide an experimental evaluation on a realistic domain.

Finite state machines become easily impractical when the plan is large and the number of actions and agents increases. To make sure our framework is applicable in complex robotic tasks we address the representation problem, and develop a formalism specifically tailored for learning in parallel or multi-agent systems. The third contribution is a novel way of using Petri Nets in learning. Memory, full parallelism among actions, and compact representation of the aspects of the state space that must be taken into account by the controller (in a way that recalls *factored* MDPs) allow for the definition of an extremely powerful tool for representation and learning.

The fourth and last contribution is an experimental evaluation of the traditional Q-learning, Sarsa(0), Sarsa(λ), and SoSMC, on non-Markovian domains. We start with two grid worlds that are quite popular in the literature of partially observable domains, and then perform a thorough evaluation on Keepaway (Stone et al, 2005), a domain whose size is more realistic with respect to real-world applications. We show that our algorithm can learn (therefore solving the *credit assignment* problem) in NMDPs, and also on multi-agent systems. Moreover, we show behaviours based on very little knowledge that perform better (collect more reward) than the complex solutions presented in the literature. This result makes us claim that enriching the representation, with the intention to let the algorithm make more informed choices, and the system to be Markovian, might be counterproductive. While RL on MDPs is proved to converge to the optimal policy, we demonstrate that such a policy is not reached in any *reasonable* time, by showing a policy that gets more reward and can be expressed in the state representation chosen. A simpler representation, requiring less knowledge of the domain and an appropriate method for learning, can speed up the process, while the optimality given up is more of a theoretical concern than a practical limitation.

Summary

The dissertation is organised in three parts.

Part I provides the essential background and introduces the notation. The work most closely related to our own is described, pointing out the respects in which our work differs. In particular, Chapter 1 introduces the basic concepts of Reinforcement Learning, and discusses the different models of Markovian, Semi-Markovian, and Non-Markovian Decision Processes. Chapter 2 gives an overview of the formalisms for plan representation, and briefly introduces Petri Nets. It also describes the two formalisms based on Petri Nets most related to our framework.

Part II provides the main contributions of this dissertation. Chapter 3 introduces the learning framework and how to derive a controllable stochastic process from a partially specified plan. Chapter 4 faces the problem of learning a controller for such a process, introducing an algorithm for non-Markovian domains. Chapter 5 addresses the representation problem, giving a novel, model-free, way of learning in Petri Nets, overcoming the limitations of finite state machines.

Part III provides an experimental validation of the framework developed throughout the dissertation, comparing the results with previous, direct, RL methods. Chapter 6 illustrates the behaviour of our algorithm in simple grid worlds, giving an insight on non-Markovian domains, and underlining the characteristics of SoSMC. Chapter 7 introduces a more complex domain, in which the framework can be fully evaluated in both the single and multi-agent case.

Part I

Background and Related Work

1

Reinforcement Learning

Reinforcement Learning (RL) is a learning paradigm which has led to a set of techniques and algorithms for solving sequential decision making problems. In such problems, the *decision maker* (or *agent*) has to face a sequence of choices, and the decisions at a certain time depend on those made in the past. Time is a fundamental aspect of RL, which poses it in between supervised and unsupervised learning. A single signal, called *reward*, is provided to the agent to evaluate its behaviour in order to maximise the performance. The problem of associating the reward to the action that has caused it is called the *credit assignment* problem, and is one the main issues that RL aims at solving. Before becoming more specific on the models and algorithms developed in RL, we focus on the principles that a few decades of research have established as the foundation of the paradigm (Sutton and Barto, 1998):

- **The reward hypothesis**: all of what we mean by goals and purposes can be well thought of as maximisation of the expected value of the cumulative sum of a received scalar signal (reward).
- **The agent hypothesis**: for the purposes of artificial intelligence, psychology, control theory and related fields, the universe is well thought of as consisting of exactly two subsystems that exchange signals over time, where the

signals in one direction are thought of as choices and the signals in the other direction are thought of as informing the choices.

- **The value-function hypothesis**: all efficient methods for solving sequential decision problems determine (learn or compute) value functions as an intermediate step. Value functions summarise the knowledge of the agent in terms of reward, they provide an estimate of the reward the agent can expect from each situation it can be in.
- **The empirical knowledge hypothesis**: (all) world knowledge is translatable, without loss of meaning, into statements about, and comparable with, future lowest-level sensations and actions.

Interaction with the environment, trial and error, self-evaluation, prediction, verification, experience, sampling, are all key aspects of an RL agent. The balance between short-term goals (immediate rewards) and long-term ones (cumulative expected reward) is expressed by the value function, that summarises the current knowledge of the agent.

Such principles have mainly been grounded in the context of Markov Decision Problems, a formulation of sequential decision making based on the model of Markov Decision Processes (MDP). The theory of Markov Decision Processes has been thoroughly developed (Puterman, 1994; Bertsekas and Tsitsiklis, 1996), and dynamic programming (DP) techniques to solve MDPs are highly correlated to the algorithms for reinforcement learning.

In the following, we briefly introduce the general framework in which most of the work on RL has been conducted. The agent is seen as the only controller of an environment it can completely perceive. The model of such an environment is an MDP that is supposed to capture all the relevant aspects of the domain, all those the agent is interested in predicting.

1.1 Markov Decision Processes

A Markov Decision Process is a tuple $MDP = \langle S, A, T, \rho \rangle$ where:

- S is a set of *states*
- A is a set of *actions*
- $T : S \times A \times S \rightarrow [0, 1]$ is the transition function. $T(s, a, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that the current state changes from s to

s' by executing action a. Since $T(s, a, \cdot)$ is a probability distribution, then $\sum_{s' \in S} T(s, a, s') = 1 \ \forall \ s \in S$ and $a \in A$. If $T(s, a, s') = \{0, 1\}$ the system is said to be *deterministic*, otherwise it is *stochastic*.

- $\rho : S \times A \times \mathbb{R} \rightarrow [0, 1]$ is the reward function. $\rho(s, a, r) = Pr(r_{t+1} = r | s_t = s, a_t = a)$ is the probability to get a reward r from being in state s and executing action a. Analogously to the transition function, $\rho(s, a, \cdot)$ is a probability density function and $\int_{\mathbb{R}} \rho(s, a, r) dr = 1$. If the reward function is defined over a discrete subset $P \subset \mathbb{N}$, ρ is a probability distribution and the reward is said to be *deterministic* if $\rho(s, a, r) = \{0, 1\} \ \forall s \in S$, $a \in A$, and $r \in P$.

We consider the system at discrete time steps. Let $t \in \mathbb{N}$ be the current time, and s_t be the state at time t. The decision maker interacts with the environment by choosing an action a_t and perceiving the next state s_{t+1}, such that:

$$s_{t+1} \sim T(s_t, a_t, \cdot) = Pr(s_{t+1} = s' | s_t = s, a_t = a) \ \forall s, s' \in S \text{ and } a \in A$$

It also receives a reward r_{t+1}:

$$r_{t+1} \sim \rho(s_t, a_t, \cdot)$$

If a state is never left, after it is entered for the first time, it is said to be a *terminal* or an *absorbing* state. If s is a terminal state then $s_{t+1} = s$ holds almost surely given that $s_t = s$. If an MDP has a terminal state it is said to be *episodic*.

1.1.1 Policies and rewards

The behaviour of the agent is represented as a function $\pi_t : S \times A \rightarrow [0, 1]$ called a *policy*, where $\pi_t(s, a)$ is the probability of selecting action a in state s at time t. If $\pi_t(s, a) = \pi'(s, a)$ for some π' and each t, the policy is said to be *stationary*, in which case we shall omit the subscript t. If $\pi(s, a) = \{0, 1\} \ \forall \, s \in S$ and $a \in A$ the policy is *deterministic*.

A policy π and an initial state s_0 determine a probability distribution over the possible sequences ($\langle s_t, a_t, r_{t+1} \rangle$, $t \geq 0$). Given such a sequence, we define the *cumulative discounted reward* as

$$R = \sum_{t \geq 0} \gamma^t r_{t+1} \tag{1.1}$$

where $0 < \gamma \leq 1$ is the *discount factor*. If $\gamma = 1$ the reward is *undiscounted*, which is allowed only if the MDP is episodic otherwise the total reward could

diverge. The discount factor expresses the preference towards earlier rewards over later ones. When a stationary policy controls an MDP the sequence of states $s_{t+1} = P(s_t, \pi(s_t, a_t), \cdot)$ is a Markov Chain.

1.1.2 Value functions

We can now define the expected cumulative discounted reward, that is the reward expected from each state s following a policy π, as:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_{t+1} \,|s_0 = s\right] \tag{1.2}$$

where r_{t+1} is the reward in the sequence ($\langle s_t, a_t, r_{t+1}\rangle$, $t \geq 0$) generated by π. The function $V : S \to \mathbb{R}$ is called the *value function*, commonly referred to just as the V-function.

Analogously to the V-function we can define an *action-value* function, that returns the expected cumulative discounted reward of executing each action from each state. It is usually represented as $Q : S \times A \to \mathbb{R}$ and known as the Q-function. Its value is given by:

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_{t+1} \,|s_0 = s,\ a_0 = a\right]$$

The reward is accumulated by executing a in s and following π thereafter.

A Markov Decision Problem is defined by an MDP and a criterion to maximise. Assuming as the criterion the cumulative discounted reward of Equation 1.1, the problem can be rephrased in terms of the value function as determining the policy π^*such that $V^{\pi^*}(s)$ is maximum for each s. We denote such an optimal value function as V^* (and analogously Q^* for the Q-function). If we represent with Π_{stat} the set of all stationary policies, and with $\rho(s, a) \sim \rho(s, a, \cdot)$ the reward extracted according to ρ, V^* is defined as:

$$\begin{aligned} V^*(s) &= \sup_{\pi\in\Pi_{stat}} V^{\pi}(s) = \sup_{a\in A} Q^*(s, a)\ s \in S \\ Q^*(s, a) &= \rho(s, a) + \gamma \sum_{s'\in S} T(s, a, s')V^*(s')\ s \in S,\ a \in A \end{aligned}$$

Stationary Markov Decision Processes, in which neither the transition function nor the reward depends on time, always admit an optimal stationary policy. Having either V^* or Q^* it is possible to derive an optimal policy by acting *greedily*

with respect to it. A greedy policy is one that always chooses the action with the highest action-value (or the expected value of the next state, in the case of the V-function), that is:

$$\sum_{a \in A} \pi^*(s,a)\, Q^*(s,a) = V^*(s)\ s \in S$$

Moreover, for an MDP, there always exists a deterministic optimal policy. Therefore, in the following we are going to consider only deterministic policies, written for simplicity as $\pi : S \to A$.

1.1.3 The Bellman operator

The definition of V^π can be rewritten in a recursive form in which the dynamics of the system are made explicit:

$$V^\pi(s) = \rho(s,\pi(s)) + \gamma \sum_{s' \in S} T(s,\pi(s),s') V^\pi(s')\ \forall s \in S \tag{1.3}$$

This linear system of equations is called the *Bellman equation*, and can be rewritten as:

$$\mathbb{T}^\pi V^\pi = V^\pi$$

Where $\mathbb{T} : \mathbb{R}^S \to \mathbb{R}^S$ is the *Bellman operator* and is defined as:

$$(\mathbb{T}^\pi V)(s) = \rho(s,\pi(s)) + \gamma \sum_{s' \in S} T(s,\pi(s),s') V(s')\ s \in S$$

Finally, we define the *Bellman optimality operator* that constitutes the foundation of the dynamic programming algorithm for solving MDPs. The optimal value function V^* is the unique fixed-point of the following equation:

$$V^*(s) = \sup_{a \in A} \left\{ \rho(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right\},\ \forall s \in S$$

That can be written as:

$$\begin{aligned} (\mathbb{T}^* V(s)) &= \sup_{a \in A} \left\{ \rho(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V(s') \right\},\ \forall s \in S \\ \mathbb{T}^* V^* &= V^* \end{aligned}$$

The Bellman operator, and optimality operator, can be similarly defined for the Q-function:

$$\begin{aligned} (\mathbb{T}^\pi Q)(s,a) &= \rho(s,a) + \gamma \sum_{s' \in S} T(s,a,s') Q(s',\pi(s'))\ \forall\ s \in S,\ a \in A \\ (\mathbb{T}^* Q)(s,a) &= \rho(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \sup_{a' \in A} Q(s',a')\ \forall\ s \in S,\ a \in A \end{aligned}$$

Most of the current RL algorithms use an estimate of the value function to store intermediate results of the optimisation process, and make it converge to the optimal function. Once the optimal value function has been reached, as previously mentioned, a deterministic stationary optimal policy can be trivially obtained, by just choosing the action with the highest state-action value in each state.

The term *learning* usually characterises those problems in which some aspects of the environment, generally the dynamics of the system modelled by the T function, are unknown. In such cases the solution cannot be *computed*, but the system must be either *simulated* or on-line *experience* must be gathered, in order to generate traces from the underlying, unknown, MDP.

Since part of the system is unknown two aspects of the learning process become particularly relevant: evaluating the current policy, and exploring new areas of the state space. Evaluation is the key component of the methods based on value functions, therefore a huge effort has been dedicated to it, especially in the early years. In the following, we summarise the main methods for policy evaluation as they will later be used to discuss our own algorithm. RL methods are usually described as composed by *prediction* and *control*. Once the value function predicts the reward returned on the long term by an action, the decision of which action to choose is still up to the agent. Clearly if the prediction is exact the choice is trivial. During the learning process, however, the agent might favour sub-optimal choices to improve their prediction. A common and simple control algorithm is ϵ-*greedy*, which chooses the best action with probability $1 - \epsilon$, while with probability ϵ chooses an action at random. In the next section we discuss a few methods for both control and prediction, we shall then move to the case where not only the dynamics of the environment is unknown, but the state is only partially observable.

1.2 Value prediction

In this section we describe the two main algorithms for value prediction, that is, the problem to evaluate, state by state, the expected cumulative discounted reward of a specific policy.

1.2.1 Every-visit Monte Carlo

The first method consists in waiting for an episode to terminate and average the return obtained from each state. Clearly it can only be applied to episodic tasks,

which are those having an absorbing state (cf. Section 1.1). Given a sequence ($\langle s_t, a_t, r_{t+1} \rangle$, $t \geq 0$) obtained following a policy π, let t_s be the first time step at which the state s appears in the sequence. The reward collected from s onward is:

$$R_{t_s} = \sum_{t \geq t_s} \gamma^{t-t_s} r_{t+1}$$

This value is used as the *target* of the update for each state. Let $\hat{V}_t^\pi$ be the estimate at time t of the value function. Every-visit Monte Carlo is characterised by the *update rule*:

$$\hat{V}_{t+1}^\pi(s) = \hat{V}_t^\pi(s) + \alpha_t(R_{t_s} - \hat{V}_t^\pi(s))$$

for each state s in the sequence.

Monte Carlo (MC) is a *multi-step* method, in that it uses the reward from multiple time steps in the future (indeed, from the whole sequence). It relies on an *estimate* since it is sampled.

1.2.2 Temporal Difference methods

Temporal Difference methods are characterised by basing the current prediction on other predictions, the ones for the following states. The update rule, therefore, has a different target with respect to MC, namely:

$$r_{t+1} + \gamma V_t^\pi(s_{t+1})$$

At each step $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$ it performs the update:

$$\hat{V}_{t+1}^\pi(s_t) = \hat{V}_t^\pi(s_t) + \alpha_t(r_{t+1} + \gamma \hat{V}_t^\pi(s_{t+1}) - \hat{V}_t^\pi(s_t))$$

This particular case of TD, called TD(0), is a *single-step* method. The prediction for the next iteration is computed, at each step, on a sample from the reward and the current prediction for the single next state. Since it makes use of a temporary prediction it is said that the algorithm *bootstraps*. While MC looks all the way down to the end of the episode, TD(0) looks forward just one step, therefore it does not need the task to be episodic. Considering this equivalence for the value

function:

$$
\begin{aligned}
V^{\pi}(s) &= \mathbb{E}_{\pi}\left[R_t(s)|s_t = s\right] && (1.4)\\
&= \mathbb{E}_{\pi}\left[\sum_{t\geq t_s}\gamma^{t-t_s}r_{t+1}|s_t = s\right]\\
&= \mathbb{E}_{\pi}\left[r_{t+1} + \gamma\sum_{t>t_s}\gamma^{t-t_s+1}r_{t+2}|s_t = s\right]\\
&= \mathbb{E}_{\pi}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1})|s_t = s\right] && (1.5)
\end{aligned}
$$

we may say that MC uses an estimate of Equation 1.4, while TD(0) uses an estimate of Equation 1.5.

There is an intermediate view, between the one step of TD(0) and the full look-ahead of MC. It consists in averaging different look-ahead lengths by interpolating through another parameter $\lambda = [0, 1]$ between MC and TD(0). This algorithm is called TD(λ), and its variants are the most used in practise since it converges much faster than TD(0). To understand its update rule, let's first define the n-step look-ahead as:

$$R_t^{(n)} = r_t + \gamma r_{t+1} + \ldots + \gamma^{n-1}r_{t+n-1} + \gamma^n R_{t+n}$$

then we weight them with λ:

$$R_t^{\lambda} = (1-\lambda)\sum_{n>0}\lambda^{n-1}R_t^{(n)} \tag{1.6}$$

where $(1-\lambda)$ normalises the weights to make sure they sum to one. Equation 1.6 is the target of TD(λ), and substituting $\lambda = 0$ we obtain TD(0), hence the name. TD(λ) provides the basis for a form of *policy iteration* (Sutton and Barto, 1998) method know as Sarsa(λ) that applies TD(λ) to the Q-function and keeps updating the policy to be greedy with respect to the value function.

1.2.2.1 Q-learning

The two methods presented earlier estimate the value function of the current policy. There exists also an instance of TD methods that computes the optimal value function while following *any* policy. Such an algorithm is called *Q-learning* (Watkins, 1989) and updates the Q-function as follows:

$$\hat{Q}_{t+1}(s,a) = \hat{Q}_t(s,a) + \alpha_t(r_{t+1} + \gamma\max_{a'\in A}\hat{Q}_t(s_{t+1},a') - \hat{Q}_t(s,a))$$

where the presence of the $\max$ operator makes the resulting system of equations highly non-linear. Q-learning has been proved to converge to the optimal value function if

$$\sum_{t}^{\infty} \alpha_t = \infty \text{ and } \sum_{t}^{\infty} \alpha_t^2 < \infty$$

and each state-action pair is visited infinitely often.

1.3 Semi-Markov Decision Processes

Semi-Markov Decision Processes (SMDPs) are a generalisation of MDPs in which transitions can take a time longer than a single time step. The transition function is defined as $T : S \times A \times S \times \mathbb{R}^+ \rightarrow [0, 1]$ (where $\mathbb{R}^+$ denotes the set of positive real numbers) and:

$$T(s, a, s', \tau) = Pr(t_{k+1} - t_k = \tau, s_{k+1} = s' | s_k = s, a_k = a)$$

This stochastic process is called semi-Markov because at time t_k, that is when the agent has to make a choice, the future of the system statistically depends only on the current state, but at other instants it may depend also on the time elapsed since the preceding transition. The reward function must take time into account too, therefore considering only discrete time steps we indicate with $\rho(s, a, k)$ the reward obtained while executing a in s after k time steps since the beginning of a. The Bellman equation for SMDPs becomes:

$$V^*(s) = \sup_{a \in A} \left\{ \sum_{\tau=0}^{\infty} \sum_{t=0}^{\tau} \gamma^t \rho(s, a, t+1) + \sum_{s' \in S} \gamma^{\tau+1} T(s, a, s', \tau) V^*(s') \right\}, \ \forall s \in S$$

SMDPs allow the modelling of *temporally extended actions* that are the building blocks of the hierarchical methods that will be described in the next section.

1.4 Hierarchical Reinforcement Learning

The description of the domain as a single MDP can grow quite large for any problem of practical interest. There are usually two ways, not mutually exclusive, to deal with this issue: *function approximation* and *abstraction*. Function approximation (Buşoniu et al, 2010) is the representation of the value function in a form other than the tabular one (that stores a value for each state, or each state-action pair). We are not going to need function approximation in this work, so in the

following we shall focus on the second method: abstraction. Abstraction is quite a general term, and it refers to coarsening the granularity at which we consider our system. A common distinction is made between *temporal abstraction*, that is the process of considering procedures longer than one time step as if they were atomic, and *spatial abstraction*, that is the process of aggregating states simplifying their representation. In this work we are mainly concerned with the former type, as the latter affects the level of models (rather than plans) and our framework is meant to sit on top of a pre-existing planning system, which includes a model on its own.

1.4.1 Introduction to Temporal Abstractions

Temporal abstractions are characterised by temporally extended actions (or *activities*), i.e., actions that take more than one time step to complete. There are four main Hierarchical RL frameworks, two of which share the same model for the activities: HAM (Parr and Russell, 1998; Andre and Russell, 2002), ALisp (Marthi et al, 2005), MAXQ (Dietterich, 2000), and Options (Sutton et al, 1999). The models for activities are respectively state machines, Lisp programs, and *options*. In this work we propose a new framework for Hierarchical RL in partially observable domains, that is related to both HAM and ALisp. Hence, these two frameworks are briefly described in the following, in order to outline the relationship to ours.

1.4.2 Hierarchy of Abstract Machines

" A HAM is a program which, when executed by an agent in an environment, constraints the actions that the agent can take in each state" (Parr and Russell, 1998). A set of state machines encodes the program, and a machine can have four possible types of states:

- **Action** states, that specify an action to execute in the environment
- **Call** states, that execute another machine
- **Choice** states, that non-deterministically select the next state machine
- **Stop** states, that halt the execution of the machine and return control to the calling machine

A HAM is defined by an initial machine and the closure of all the machines reachable from the initial one. Together with the HAM H, it is assumed that the

description of the MDP M, that models the domain, is available. The composition $H \circ M$, that is the *application* of H on M, is obtained by: making the Cartesian product of the state spaces of H and M, and picking transitions either from the machine, if the state is an action state, or from the MDP if it is a choice state, expanding the corresponding machine otherwise. The reward is taken from the MDP if the transition is an action, while it is zero otherwise. The resulting process is still an MDP, and it is proved that there is no need to store the entire state space, since there exists always an SMDP such that its optimal policy is the same as $H \circ M$, but it has only the states $\langle m, s \rangle \in H \times M$ in which m (the machine state) is a choice state. Learning is performed at choice points, and for each transition from $\langle m_c, s \rangle$ to $\langle {m'}_c, s' \rangle$, where m and m' are both choice states, the Q-function is updated as in:

$$Q_{t+1}(\langle m_c, s \rangle, a) = Q_t(\langle m_c, s \rangle, a) + \alpha(r_c + \gamma^\tau V_t(\langle {m'}_c, s' \rangle) - Q_t(\langle m_c, s \rangle, a))$$

where r_c is the cumulative discounted reward collected during the execution of action a and τ is the time that a has taken to terminate.

The agent executes the actions specified by the machine in the underlying MDP as long as such a specification is deterministic. When a non-deterministic choice point is encountered during execution, the full MDP is used to *learn* the best choice among the ones selected by the machine in that specific state. This effectively reduces the policies available for learning to those compliant to the structure of the HAM. HAMs can be seen as the model-free counter-part of decision theoretic planning, like for instance DTGolog (Boutilier et al, 2000), although in the latter case, besides the golog program, there is also a symbolic model in the situation calculus.

HAMs are limited to single agent systems, with complete knowledge, and that can execute one single action at a time. Our work introduces a Hierarchical RL framework that can be used in domains with only partial knowledge, multiple actions, and also multiple agents. Differently from HAMs, we do not flatten the hierarchical structure all the way down to an MDP (that we do not assume to have), but learn on the hierarchy itself.

1.4.3 ALisp

ALisp is an extension of the Lisp programming language to implement the same learning mechanism as HAMs. A few new operations are added to the language to implement the non-deterministic choice, and to specify which action must be exe-

cuted on the underlying MDP. Several features have been developed over HAMs, and in particular over *programmable* HAMs (Andre and Russell, 2001). Since Lisp is a full programming language, both the variables and the program counter enrich the description of the machine's state space. Moreover, the framework has been extended to include function approximation, state abstraction (Andre and Russell, 2002), and concurrency (Marthi et al, 2005).

In the concurrent setting the state of all the threads is considered, and the global state is regarded as a choice point when at least one thread is in a choice state. The actions available at such choice states are the joint actions available to the threads at a choice point. Thus, choices are made globally and at the same time for all agents. The framework we are defining in Chapter 3 has similar global choice states, while its extension with Petri Nets in Chapter 5 allows to make distributed choices for different agents, or threads, separately.

1.5 Non-Markovian Decision Processes

Non-Markovian Decision Processes (NMDPs) are the most general class of controllable stochastic processes, in which the statistical dependency of the dynamics or the reward is not limited in any way. While in a Markov Decision Process the current state is sufficient to determine the distributions of both the next state and the reward, this cannot be assumed on NMDPs. For finite processes, that is those in which both the set of states and actions are finite, the number of possible policies is also finite, but it is exponential in the number of states. On the other hand, we know that for MDPs Dynamic Programming can find an optimal policy in time polynomial in the number of states. As stated by the *no free lunch theorem for optimisation* (Wolpert and Macready, 1997), there is no algorithm that performs better than random search on *any* possible domain. Every algorithm must then be tailored for a specific sub-class of problems, and we have seen how we can successfully solve MDPs in both the model-free (Monte Carlo and Temporal Difference methods, cf. Section 1.1) and model-based (Dynamic Programming) setting, indeed exploiting the Markov property.

Assuming the Markov property presumes that the designer has full knowledge about the environment and the agent full observability. This clearly does not mean that any of them must be omniscient. It means that the designer must be able to come up with a description of the state space that: (1) includes the aspects that the agent is interested in predicting, and (2) such that both the distribution

of the next state and the reward statistically depend solely on the current state. The agent, on the other hand, must be able to perceive those aspects at any time. This assumption is quite strong for robotic applications, especially when the decisions to be made are not at the level of joints, but involve many aspects of the environment and possibly other agents.

In order to cope with partial observability, the model of Partially Observable Markov Decision Process (POMDP) has been developed. A POMDP is a tuple $\langle S,\ A,\ T,\ \rho,\ Z,\ O\rangle$, where $\langle S, A, T, \rho\rangle$ is an underlying MDP whose current state is not directly accessible. Instead of perceiving an element from S, the agent is given an element of Z, the set of *observations*, which relates to the underlying state through the function $O : Z \times A \times S \to [0, 1]$ such that $O(z, a, s) = Pr(z|s, a)$ is the probability of observing z when executing a in s. Most of the methods for dealing with POMDPs take advantage of some form of memory, often implemented as *belief states*. A belief state is a distribution over the possible states of the MDP, and summarises the agent's *belief* about where the past actions might have taken it. The process built on top of belief states is still an MDP, but its complexity is prohibitive, since the space of all possible distributions is continuous even if the state space is discrete. If the distribution is ignored, and we pose the problem of controlling the system with a memory-less policy on the observations, the resulting process is not Markovian anymore. This problem is similar to the one we are considering, but we move a step further in the knowledge assumptions. POMDPs still assume that a Markovian description of the state space is available. In this dissertation we consider the implications of having only a partial specification of the state space, that is, a model with partial knowledge for which we cannot assume the Markov property. Such an assumption leads to the general case of Non-Markovian domains.

The literature about RL methods in Non-Markovian domains is far less rich than for MDPs, and less well organised. In the following, we gather the results about such settings, and summarise the current understanding about what RL methods learn when the Markov property is not guaranteed. Most of the work on the subject has been carried out assuming a POMDP setting, which we can consider acceptable if no use is made in the algorithms of both the dynamics of the underlying MDP and the definition of the state space, other than to set up the theoretical framework.

1.5.1 Policies for POMDPs

Singh et al (1994) showed how, despite for MDPs there always exists an optimal deterministic stationary policy, this is not true for POMDPs. They point out that in a POMDP:

- just confounding two states of an MDP *can* lead to an arbitrary high absolute loss in the return, or cumulative infinite discounted payoff
- the best stationary stochastic policy *can* be arbitrary better than the best stationary deterministic policy
- the best stationary stochastic policy *can* be arbitrary worse than the optimal policy in the underlying MDP
- the optimal policy *can* be non-stationary
- there need not be a stationary policy that maximises the value of each observation simultaneously

Despite these negative results, we are interested in deterministic stationary policies for they are computationally more appealing, and because there are several experimental results in which a simple, memory-less, deterministic policy can perform little worse than the optimal one. Also the examples used to prove the former sentences have been tailored specifically, and there is a chance that real problems do not have the characteristics of the worst cases. This is often true with NP-complete problems, where SAT solvers can solve industrial problems with thousands of variables, despite the existence of intractable instances.

1.5.2 Value prediction in NMDPs

The value of an observation depends on the underlying state and, every time an observation is obtained, the reward will correspond to the one returned by the hidden state associated to the observation. Given the definition of the value of a state s under a policy π of Equation 1.3, reported in the following for ease of reference

$$V^{\pi}(s) = \rho(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V^{\pi}(s') \; \forall s \in S$$

the value of an observation o under π depends on the probability with which each state can be the one underlying o when following π. Singh et al (1994) define a

function $Pr^{\pi}(s|o)$, calling it the *asymptotic occupancy distribution*, as :

$$Pr^{\pi}(s|o) = \frac{Pr(o|s)Pr^{\pi}(s)}{Pr^{\pi}(o)} = \frac{Pr(o|s)Pr^{\pi}(s)}{\sum_{s' \in S} Pr(o|s')Pr^{\pi}(s')} \; \forall s \in S, o \in O$$

where $Pr^{\pi}(s)$ is the limiting distribution over the hidden state space and is well defined under the assumption that such an MDP is ergodic under any stationary distribution, that is, the corresponding Markov chains are aperiodic, and the expected time for each state to appear again in the chain is finite. They then define the value function for POMDPs in terms of Pr^{π} as:

$$V^{\pi}(o) = \sum_{s \in S} Pr^{\pi}(s|o)V^{\pi}(s) \; \forall o \; \in O \tag{1.7}$$

The authors also prove what TD(0) and Q-learning converge to in such a setting. TD(0) converges with probability one, under the same conditions required on MDPs plus the condition that the learning rates (α) are non-increasing, to the solution of the following system of equations:

$$V(o) = \sum_{s \in S} Pr^{\pi}(s|o) \left[\rho(s, \pi(o)) + \gamma \sum_{o' \in O} \sum_{s' \in S} T(s, \pi(o), s')Pr(o'|s')V(o') \right] \; \forall o \in O$$

such a solution does not necessarily correspond to the desired value function of Equation 1.7. As for Q-learning, since the policy used while collecting data impacts the occupancy distribution, it is not possible to retain its off-policy behaviour (learning the optimal value function while following any policy). Under one more assumption, namely that the policy followed assigns a non-zero probability to every action in every state, Q-learning converges with probability one to the solution of the following system of equations, $\forall o \in O, a \in A$:

$$Q(o, a) = \sum_{s \in S} Pr^{\pi}(s|o, a) \left[\rho(s, a) + \gamma \sum_{o' \in O} \sum_{s' \in S} T(s, a, s')Pr(o'|s') \max_{a' \in A} Q(o', a') \right]$$

This solution has the same problems as for TD(0) because of the 1-step Markov assumption, and the fact that it searches for a deterministic policy.

1.5.3 Learning equilibria

In Markov Decision Processes sub-optimal policies are unstable under policy iteration, so that each step produces not only a different policy, but a better one. MDPs are, therefore, well suited to hill-climbing, since all the optima form a single, connected, plateau. Optimal policies, and only those, are equilibrium points

in MDPs, while in NMDPs it is possible that a sub-optimal policy is an equilibrium point for policy iteration. Pendrith and McGarity (1998) define a class of POMDPs in which the history is enough to determine the state (hPOMDP, where "h" stands for history) and analyse the stability of TD(λ) (that was missing in the work described in the former section) and first-visit MC. Given a sequence ω = ($\langle s_t, a_t, r_{t+1} \rangle$, $0 \leq t \leq n$) the reward accumulated during the sequence is:

$$R(\omega) = \sum_{t=0}^{n} \gamma^t r_t$$

The authors define the value of a policy π as:

$$J(\pi) = \int_{w \in T} R(\omega) dPr^{\pi}(\omega)$$

where T is the set of all possible traces, and $Pr^{\pi}(\omega)$ is the probability of the policy π to generate the trace ω. In this setting, they prove the following results:

- if a first-visit MC method of credit assignment is used for an hPOMDP where $\gamma = 1$, then the optimal observation-based policies will be learning equilibria.
- the previous result does not apply to $\gamma = [0, 1)$
- if a TD(λ) method of credit assignment is used for direct RL on a NMDP, then for $\gamma < 1$ it is not guaranteed that there exists an optimal observation-based policy representing a learning equilibrium.

Perkins and Pendrith (2002) carry this analysis further, and include the exploration policy explicitly. They prove that there exists a learning equilibrium for 1-step TD methods if the exploration policy is *continuous* in the action values, while most of the former analysis had been conducted with ϵ-greedy which is discontinuous. So, for instance, following SoftMax, that assigns to every action a probability according to a Gibbs, or Boltzmann, distribution:

$$Pr(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A} e^{Q(s,a')/\tau}} \tag{1.8}$$

both Sarsa and Q-learning have at least one action-value function that is a learning equilibrium, i.e., a fixed point for

$$Q = Q_{Sarsa}(Softmax(Q))$$

and

$$Q = Q_{Q-learning}(Softmax(Q))$$

respectively, where Q_U is the update rule that uses method U, and $Softmax$ is the function that selects an action according to the probability in Equation 1.8. The parameter τ in 1.8 balances exploration and exploitation: the highest τ the more the agent is likely to select a sub-optimal action (according to the *current* value function). The results presented prove that there exists a fixed point with respect to the update rule and a continuous exploration policy, but do not prove that such a fixed point can actually be reached. Moreover, the presented results do not consider that the exploration may change, for instance letting τ tend to zero.

Summarising, first-visit Monte Carlo and TD in the undiscounted case admit a learning equilibrium for their optimal observation-based policy. Moreover, TD admits at least one equilibrium in the discounted case under a continuous strategy for action selection, if the hidden MDP is ergodic.

1.5.4 A theoretically sound algorithm for local searching

The work on fix points has inspired a theoretically sound algorithm for POMDPs based on ideas of stochastic search, while retaining an RL-style update. Perkins (2002) redefined the value function to overcome the aforementioned difficulties with discounted problems.

Let $\mu(\pi)$ be the probability distribution over the possible trajectories determined by a policy π. The author splits the reward with respect to an observation o from one of these trajectories ω in:

$$\begin{aligned} V^\pi &= \mathbb{E}^\pi_{\omega\sim\mu(\pi)}\left[R(\omega)\right] \\ &= \mathbb{E}^\pi_{\omega\sim\mu(\pi)}\left[R_{pre-o}(\omega)\right] + \mathbb{E}^\pi_{\omega\sim\mu(\pi)}\left[R_{post-o}(\omega)\right] \end{aligned} \tag{1.9}$$

where $R_{pre-o}(\omega)$ is the cumulative discounted reward before o is encountered in ω for the first time, while $R_{post-o}(\omega)$ is the reward after the first occurrence of o. In the following, we shall omit the subscript $\omega \sim \mu(\pi)$, but all traces must be intended to be extracted from $\mu(\pi)$ if not otherwise noted. The value of an observation-action pair $\langle o, a\rangle$, with respect to a policy π, is the value of the policy when π is followed everywhere except for o, in which a is executed instead. Such a policy is represented as $\pi \leftarrow \langle o, a\rangle$, and clearly $\pi = \pi \leftarrow \langle o, \pi(o)\rangle$. Its value is:

$$Q^\pi(o, a) = \mathbb{E}^{\pi\leftarrow\langle o,a\rangle}\left[R_{post-o}(\omega)\right] \tag{1.10}$$

This definition differs from the usual definition for MDPs, given in Equation 1.2, in two important respects: (1) every time (and not just the first one) the observation o is encountered, the agent executes the action a; (2) the value of an observation-action pair is not the discounted return following o, but the expected discounted reward following o *in that point* of the trace. Since an observation can happen at different times, in different positions of the trace, the discount factor in $R_{post-o}(\omega)$ is different depending on this time.

While in MDPs the optimal policy is greedy with respect to the action-value function, as mentioned in Section 1.5.2, this is not necessarily true for POMDPs. With the definition of the value function just given, this property is retained to some extent. In particular, it is proved that given two policies π and $\pi' = \pi \leftarrow \langle o, a \rangle$:

$$V^{\pi}(o) + \epsilon \geq V^{\pi'}(o) \Longleftrightarrow Q^{\pi}(o, \pi(o)) + \epsilon \geq Q^{\pi'}(o, a)$$

MCESP is a family of algorithms that make use of the value function of Equation 1.9 to compute a *locally optimal* policy, updating one observation-action pair at a time. At the beginning of each trial it chooses a policy $\pi \leftarrow \langle o, a \rangle$ that is used to generate a trajectory. At the end of the episode the value $Q(o, a)$ is updated based on R_{post-o} and the algorithm checks whether the current policy should change. The scheduling of the constants (α, ϵ), the way in which the observation-action pair to explore is chosen, and the conditions under which the procedure terminates, determine different algorithms that incorporate various ideas from RL and stochastic search.

Although the gained capability to hill-climb guarantees at least a local optimality, updating one state-action pair at a time, and spending a few episodes per attempt to evaluate the new policy, make MCESP quite slow. To address this issue, in Chapter 4 we define a new algorithm for stochastic search that retains some of the ideas behind MCESP, while attempting a biased global search. Our algorithm relies on the ability of MC policy evaluation to estimate the current policy, and performs a form of branch and bound related to *confidence bounds* (Auer, 2002; Auer et al, 2009) for NMDPs.

2

Plan Representation and Petri Nets

The frameworks for plan representation and execution (while generation is a different subject) can generally be categorised into three approaches, distinguished by the formal model they are based on: Finite State Automata (FSA), programming languages, and Petri Nets (PN). From a theoretical point of view, if we do not limit automata to be finite, the three models are equivalent (can simulate one another). Petri Nets are strictly more expressive than FSA (Murata, 1989), and have a broad tradition as a modelling tool for engineering and manufacturing applications (Viswanadham and Narahari, 1992; van der Aalst, 1998). The models we are mainly interested in are FSA and PN, since we are going to develop our hierarchical framework for both. In the following, we shall first give an overview of the current systems using both models, and then introduce the closest formalisms to the one developed in this work.

2.1 Overview of FSA and PNs for plan representation

Most robot programming languages are based on Finite State Automata (FSA). FSA are either used explicitly, possibly supported by a graphical language, or

they provide the underlying semantic model for the language.

Colbert (Konolige, 1997) is a robot programming language which was developed as a component of the Saphira architecture (Konolige et al, 1997). Despite the fact that Colbert has a syntax which is a subset of ANSI C, its semantic is based on FSA. In particular, states correspond to actions while edges are events associated to conditions. Moreover, Colbert allows some simple form of concurrency although, in this case, the semantics is considerably different from standard FSA semantics and it is very hard to guarantee coherence in the behaviours. Probably, the most interesting feature associated to concurrency in Colbert is the possibility to monitor and interrupt actions.

The limitations of FSA have raised the issue of finding more expressive formalism to control robots. Some approaches, such as ESL (Gat, 1992), address the problem by defining constructs commonly used in robotics, without limiting the expressiveness of the programming language which is based on Lisp. In a similar way, the Task Description Language (TDL Simmons and Apfelbaum, 1998) extends C++ in order to include asynchronous constrained procedures, called Tasks. TDL programs have a hierarchical structure, called Task Tree, where each child of a given task is an asynchronous process and execution constraints among siblings are explicitly represented. Xabsl (Loetzsch et al, 2006) is a more recent approach, mainly developed in the frame of the RoboCup competition and is somehow similar to TDL. This approach is based on a hierarchical structure and is bundled with a set of highly engineered tools which allow efficient development of behaviours.

The Reactive Action Packages (RAPs, Firby, 1989) is the robot programming language of the Animate Agent Architecture (Firby et al, 1998). RAPs are expressed in Lisp-like syntax and describe concurrent tasks along with execution constrains. RAPs are an ad-hoc tool for the execution of concurrent tasks in robotic applications which have some similarities with PNs. Nevertheless, it is not possible to perform analysis of RAPs, mainly because there is no underlying formal model.

Although FSA-based approaches have been very successful in modelling many single-robot systems, their expressive power limits their applicability to multi-robot ones. In general, more expressive formalisms are required in order to model the inherent concurrency of multi-robot systems. The extensions of FSA-based approaches to handle concurrency usually have ad-hoc semantics and do not allow formal analysis, making it difficult to develop robust and effective behaviours.

In the past few years, approaches to plan generation and representation based

on Petri Nets have gained increasing interest. PNs provide a mathematical and graphical framework for the representation of discrete event systems. An advantage provided by PNs to this extent is the possibility to provide formal properties of the produced models, using some standard analysis methods.

Petri Nets have been used for both single and multi-agent systems. Celaya et al (2007) present a framework to build multi-agent systems using Petri Nets, that allows to perform static analysis to assess some important properties of the system, for instance for deadlock avoidance. The model is limited to purely reactive agents: actions are instantaneous, represented by Petri Net transitions, and the places of the Petri Net model represent the environmental state of the agent. The assumption of instantaneous actions does not allow to adequately model robotic systems where actions have a duration and must be monitored during execution. Similar considerations apply to the work by Best et al (2001), where the authors present a Petri Net Algebra, which allows the design of multi-agent systems (MAS) using a component-model approach, based on composition operators. In this approach, individual and system goals are specified as reachable markings in the Petri Net representing the MAS. Interestingly, the proposed methodology is property-preserving, and ensures that formal characteristics of the net are maintained in the composition of the multi agent system.

There has been other work which addresses specifically robotic systems. Nevertheless, these approaches develop ad-hoc models for specific applications, rather than providing a general language. For example, Sheng and Yang (2005) use PNs to model a multi-robot coordination algorithm, based on an auction mechanism, to perform environment exploration. Similarly, Xu et al (2002) show an agent-based extension of Fuzzy Timed Object-Oriented Petri Nets (proposed by Maier and Moldt (2001)) for the design of cooperative multi-robot systems for a specific industrial application. Moreover, Kuo and Lin (2006) reports the use of distributed agent-oriented Petri Nets for the modelling of a multi-robot system for soccer playing.

King et al (2003) provide a general approach able to model multi-agent systems, in which plans for each single robot are generated either using a graphical interface, or using some automated planning method. The plans are then compiled into Petri Nets for analysis, execution, and monitoring of their joint execution. The operators that are used for the PN representation of the plans are inspired by the STRIPS Fikes and Nilsson (1971) planning system. Supervisory control techniques are applied to the Petri Net controller in order to identify

possible conflicts that may arise due to the presence of shared resources among the multiple robots. To deal with unforeseen events re-planning is used at run-time severely limiting the applicability of this approach to real-rime systems in dynamic environments.

Milutinovic (2002) proposes an approach for modelling single-agent systems extended in several respects, including multi-agent domains, by Costelha and Lima (2007). The designer provides a layered model of the environment, the actions available to the agent, and their interactions, then the system composes those models into a Petri Net used for analysis. The resulting process is an MDP, whose evolution can be studied with the methods of Chapter 1, and that provides a model-based approach for decision making. As already discussed, writing a complete and reliable model for a robotic domain is particularly hard, so we prefer an approach closer to the one by Ziparo et al (2010), where the agent's behaviour is modelled but the environment is not. Both have their merits and drawbacks, so before developing a system that contributes to overcome some of the disadvantages in both cases, these two formalisms will be introduced in the next sections.

2.2 Petri Nets and related formalisms

The two approaches most closely related to our work are those by Ziparo et al (2010) and by Costelha and Lima (2007). After a brief introduction about Petri Nets to clarify the notation, we are going to introduce those two formalisms and discuss their features and how they are related to our own work.

2.2.1 Introduction to Petri Nets

Petri Nets (Murata, 1989) are directed, weighted, and bipartite graphs. A Petri Net has two types of nodes connected by directed weighted arcs (which defaults to one if not otherwise specified). Nodes of the first type are called *places* (Fig. 2.1(a)) and may contain zero or more *tokens* (Fig. 2.1(c)). The number of tokens in each place is called *marking*, and denotes the state of the system.

Nodes of the second type, called *transitions* (Fig. 2.1(b)), represent the events modelled by the system.

A Petri Net can be defined as a tuple

$$PN = \langle P, T, F, W, M_0 \rangle$$

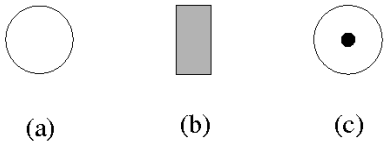


Figure 2.1: Nodes of a Petri Net

where:

- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of *places*.
- $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of *transitions*.
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of edges.
- $W : F \rightarrow \mathbb{N}^+$ is a weight function and $w(n_s, n_d)$ denotes the weight of the edge from n_s to n_d.
- $M_0 : P \rightarrow \mathbb{N}^+$ is the initial marking.
- $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$

Transitions can produce or consume tokens from places according to the rules defining the dynamic behaviour of the Petri Net. The dynamic behaviour of the network, that is how the state changes, is defined by the following *firing rule*

1. A transition t is *enabled*, if each input place p_i (i.e. $(p_i, t) \in F$) is marked with at least $w(p_i, t)$ tokens.

2. If an enabled transition t fires, $w(p_i, t)$ tokens are removed for each input place p_i and $w(t, p_o)$ are added to each output place p_o such that $(t, p_o) \in F$.

There are different semantics for Petri Nets depending on how and when an enabled transition is chosen for firing. If the particular semantic should be relevant to a specific application, it will be clarified when necessary.

2.3 Petri Net Plans

Petri Net Plans (PNPs) are particular Petri Nets whose operational semantics is enriched with the use of conditions verified at run-time over an external Knowledge Base. No restriction is imposed on the KB, which is supposed to be updated by other modules according to the agent's perceptions.

(a) Ordinary Action (b) Sensing Action

Figure 2.2: Ordinary and sensing actions in a Petri Net Plan

The basic actions of a PNP are *ordinary actions* (see Figure 2.2(a)), which represent a durative action, and *sensing actions* (figure 2.2(b)), representing procedures whose outcomes reflect the truth of one or more conditions (equivalent to an if-statement). Places and transitions of a PNP are partitioned into sets that represent specific aspects of the plan:

- *Input* places model initial configurations of the network, before the plan has been executed.
- *Execution* places model the configurations during which actions (and sub-plans) are executed.
- *Output* places represent the time after the execution has terminated
- *Connector* places are used to compose networks by means of operators

In the example of Figure 2.2 the input, execution, and output places can easily be picked out for the small networks that represent single actions. A sensing action has more than one output place, having more than a possible outcome.

Transitions are partitioned as well, in particular:

- *Start* transitions represent the beginning of an action or a plan.
- *Termination* transitions model the end of an action or a plan
- *Control* transitions, as well as connector places, are used in operators

A Petri Net Plan is the composition of the aforementioned basic actions by means of a set of operators. We introduce the operators informally and refer to the original paper (Ziparo et al, 2010) for a formal definition:

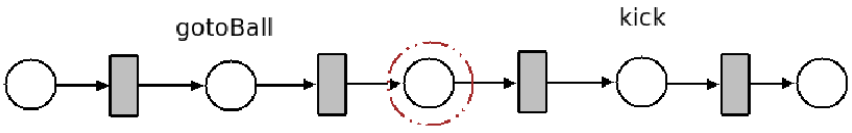


Figure 2.3: The `sequence` operator

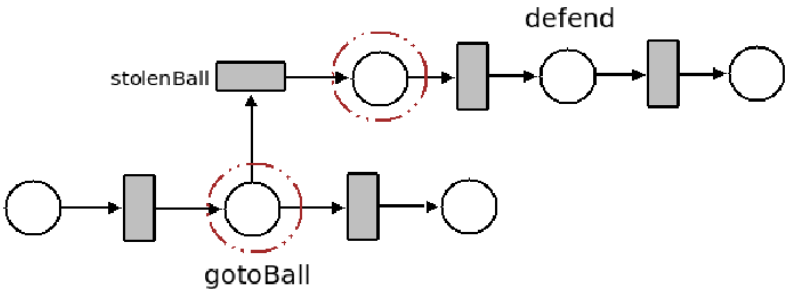


Figure 2.4: The `interrupt` operator

- *Sequence*. A sequence is obtained by merging two places of different PNPs. It can be applied to any place but execution ones; an example is provided in Figure 2.3
- *Interrupt*. Interrupts connect the execution place of an action (or a sub-plan) to a non-execution place of another network. It causes the temporary termination of such an action under anomalous circumstances, that is, other than those accounted for by terminating transitions. An example is provided in Figure 2.4
- *Fork* and *Join*. Each token in a Petri Net Plan can be thought of as a thread in execution. Through `Fork` and `Join` threads can be created and synchronised, as usually done in most programming languages. An example of a fork followed by a join is shown in Figure 2.5. Interrupts are often used to repeat a portion of a plan that did not realise its post-conditions, implementing while-loops.

There are also operators for multi-agent plans but for now we limit ourselves to single-agent plans.

Summarising, a Petri Net Plan is formally a tuple

$$PNP = \langle P, T, W, M_0, G \rangle$$

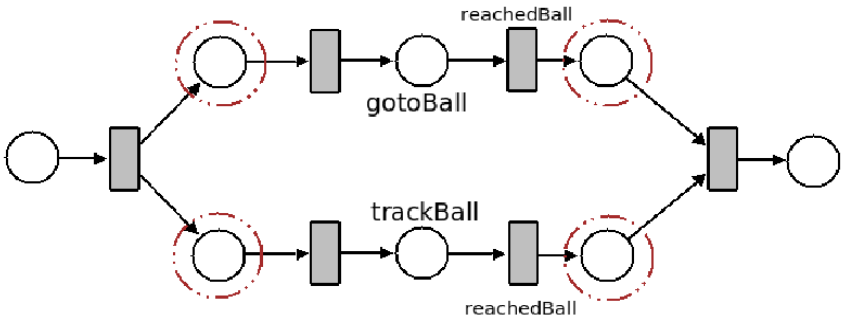


Figure 2.5: The `fork` and `join` operators

in which $\langle P, T, W, M_0 \rangle$ is a Petri Net, and where:

- P is a set of *places* that represent the execution phases of actions. Each action is described by three places: the *initiation*, *execution*, and *termination* place. Those places denote respectively the time before the action starts, during its execution, and after the action terminates.

- T is a set of transitions that represent events and are grouped in different categories: action-starting transitions, action-terminating transitions, interrupts, and control transitions (which are part of an operator). Transitions may be labelled by conditions to control their firing.

- $W : T \rightarrow \{0, 1\}$ is the weight function. In PNP an arc either does not exist (zero weight) or has weight one.

- G is a set of *goal* markings that indicate the termination of the plan. It must be a proper subset of the possible markings in which a PNP can be.

Moreover, it must have the following characteristics:

Definition 1 (Safety)**.** *A PNP is safe if any reachable marking M satisfies:*

$$M(p) \leq 1 \; \forall p_i \in P$$

In Petri Net terms the network is said to be *1-bounded*.

Definition 2 (Minimality)**.** *A PNP is minimal if every transition is in at least one possible firing sequence from the initial marking.*

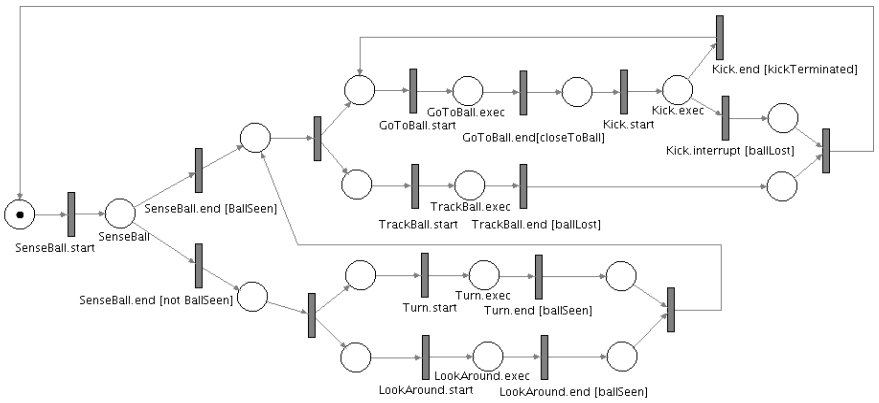


Figure 2.6: A Petri Net Plan for a football playing robot

In Petri Net terms the network is said to be *L1-live* (where from L1 to L4 transitions range from being possibly fired to being fired infinitely often). These two structural properties are decidable and verifiable trough standard reachability and liveness techniques for PNs. Finally, Petri Nets, in order to be PNPs, must have one last property, due to their interpretation as plans:

Definition 3 (Effectiveness)**.** *A PNP is effective if every goal marking is reachable from any state.*

As an example of a Petri Net Plan, consider the behaviour in Figure 2.6 which is a simple plan for a football playing robot. The plan starts with a sensing action to check whether the agent knows the position of the ball or not. This determines two branches (but still one token), each of which has two actions executing in parallel. Therefore, the `fork` operators double the number of tokens, that are reconciled by their respective `join` if the control must leave the parallel section. The action `Kick` can be interrupted in two different ways: by its normal termination condition, that is when the movement has finished, or by the undesirable condition of `ballLost`, that is when the ball is not in the robot's sight anymore. In this particular case the goal set is empty, as the behaviour keeps going indefinitely.

Petri Net Plans are particularly effective in representing the constraints among actions in parallel or multi-agent plans (in which case all actions have a further label that denotes the agent in charge of executing the action). PNPs are just plans, and there is no element modelling the environment. Sensing actions bridge the gap between the agent's beliefs and the action execution. In terms of learning, using conditions to determine the state before each decision can result in unman-

ageably large plans. In Chapter 5, we extend PNP for learning and show how to exploit PNs' expressive power for representing memory and compactly factorise the state space.

2.4 A model-based approach with Petri Nets

Costelha and Lima (2007) propose a model-based framework for modelling the environment and the agent behaviour in a layered structure. The lowest layer represents the *environment*, and has a place for each predicate describing the state space. A token in a place is interpreted as the corresponding predicate being true, and transitions impose constraints on the possible evolutions of the state. The second layer is the *action* layer, in which each action is separately modelled with pre, execution (conditions that must hold for the action to keep executing), and post-conditions. A place is also dedicated to represent the execution status of each action, so that when there is a token in it the action is considered active. Such a place is the conjunction with the above layer, the layer of plans, or *action coordinator*. In this layer actions are composed to form plans and behaviours. The topmost layer is the *coordination* one, in which the agents are assigned to the tasks. Once the modelling is completed, all the Petri Nets representing the various elements of the system are merged, into one single network that can then be analysed as an SMDP with formal tools for Timed Petri Nets.

The approach described requires a deep knowledge of the domain, and quite some confidence on action models. Since we argue that this is difficult to achieve in complex tasks, we prefer to use the procedural approach of PNP, and combine it with perceptions at run-time and decisions learnt for experience, creating a model-free framework.

2.5 Summary of related work

The analysis of related work moves along three different lines: Hierarchical RL, algorithms for non-Markovian domains, and representation and learning agent behaviours with Petri Nets.

For what concerns HRL, we are going develop a framework for parallel execution under partial knowledge suitable for both single and multi-agent systems. The closest formalism in the literature are HAM and ALisp (cf. Section 1.4), which are limited to the case of full observability. HAM is also limited to a single action

at a time, while ALisp has been extended for multi-threaded execution, although the choices are made globally to retain the Markov property. Both frameworks assume the existence of an MDP and the description of the state space to be available. On such a description, they flatten the structure imposed by the hierarchical state machine and obtain an SMDP that can be used for learning. Since we do not assume to have a Markovian description of the state space, we retain the hierarchy during learning, and decompose the value function accordingly.

Since the stochastic process whose controller we learn is not Markovian in general, we have followed the research line about such domains and gathered the results obtained so far in Section 1.5. A difficulty for direct RL on NMDPs is that the best deterministic policy does not necessarily maximise the value function in each state. With a different definition of the value function, though, this limitation can be overcome subject to local searching. We connect a few ideas from MCESP (the only sound algorithm for local search in NMDP), the work on consistent exploration, confidence bounds, and stochastic optimisation, we devise an algorithm for stochastic global search in policy space for non-Markovian environments.

In order to make our framework easier to use on robotic systems, after having developed our argument for finite state machines, we extend it to Petri Nets, whose expressive power is particularly helpful in parallel and multi-agent systems. We provide a brief survey of the available formalisms in Section 2.1, and then discuss Petri Net Plans (cf. Section 2.3) that define a language for representing complex single and multi-agent plans, and a model-based approach for the analysis of behaviours with PNs (cf. Section 2.4). The former turns out to be not suitable for learning as it is, while the latter cannot be use with partial knowledge. In order to address both those issues, we extend PNP and define a novel way for learning agent behaviours with Petri Nets.

Part II

Combining Planning and Learning

3

From Plans to Controllable Stochastic Processes

Our approach addresses the planning problem in complex domains, where the agent is not the only one affecting the environment. A complete description of the causes of state changes is generally not available, and several aspects of the environment may be not observable. We therefore focus on the only aspect that is always available: the plan. We do not want to limit the expressiveness of plans we consider in any way, so we allow hierarchical plans with parallel action execution, sensing, loops, and interrupts. We are going to develop our argument for State Charts (Harel, 1987) first, to which every formalism equivalent to a finite state machine can be cast, and then address the expressiveness in case of multiple actions executable at the same time.

3.1 Plan Representation

We consider reactive plans represented as generic state machines, like state charts (Harel, 1987), in which every state corresponds to a set of *actions* and each transition corresponds to an *event*. An action may also be a machine itself. Figure 3.1 shows a simple machine defining the overall behaviour of a football playing

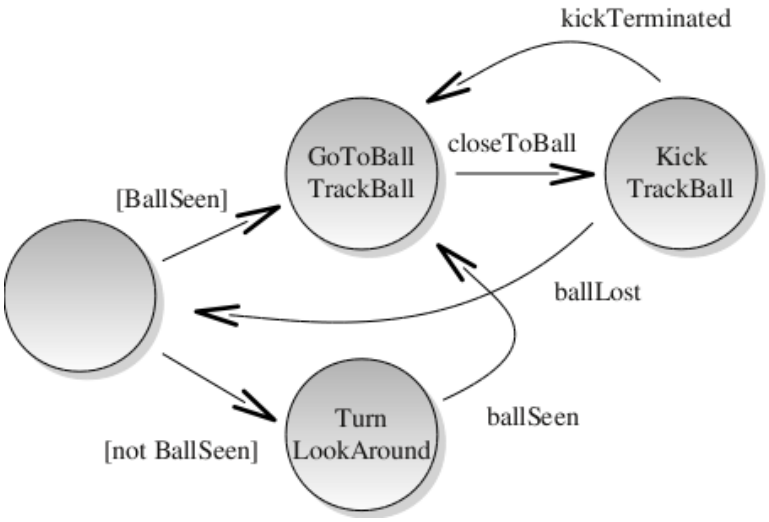


Figure 3.1: A simple behaviour for a football playing robot

robot, as those used in the RoboCup Soccer (Kitano et al, 1997). This example will be further developed later.

3.1.1 The main components

A *plan state* is a configuration of the machine that encodes the plan, as opposed to an *environment state* that is a configuration of the domain independent from the agent. Each plan state is associated the set of *actions* that is being executed at that point of the plan. Notice that the same set of actions may occur several times in different plan states. The state of the whole system is the Cartesian product of the plan and the environment state spaces. Such a product cannot be computed though (and doing so is probably not even desirable), since a description of the external environment is not generally available. In our example, every node is a plan state and each state is associated with two actions (except for the first state, that will be explained soon), one to control the head of the robot and one for the body. The two actions must be executed in parallel and need not be atomic, but might be time-extended procedures therefore machines themselves. During the execution of an action, the environment state changes continuously while the plan state does not. Indeed, this representation does not model explicitly the agent's knowledge, but only the execution state of the plans.

The outcome of actions may be uncertain, and we assume that a knowledge base (KB) exists such that at any moment it is possible to check whether or not it

entails a certain condition. We also assume that appropriate modules keep the KB updated with respect to the agent's perceptions. Such a knowledge representation interferes in the plan execution in two different ways: events and conditions.

An *event* is a happening in the environment that the agent is capable of detecting, for instance: a condition that becomes true, a message received from another agent, a timeout expired or a joint that reached its target position. Events mark perceivable changes that are particularly relevant, among all the possible perceptions, since they trigger a change of the current state. Events model the conditions under which a set of actions is considered successful and at least some, of the generally many, sources of failure. For instance, in Figure 3.1 *closeToBall* is the event detected when the agent gets nearer to the ball than a certain threshold, and is the correct termination of the set of actions {GoToBall,TrackBall}. On the other hand, *ballLost* is the event detected when the ball is not on the robot's sight any longer, and is an incorrect termination for the actions {Kick,TrackBall}. There is no difference, at the plan level, between *ballLost* and *kickTerminated*, where the latter is the correct termination of that set of actions. We do not impose any semantic interpretation to the symbols and it is the responsibility of the planner to make sure that the plan is correct.

In addition to events, edges are labelled with *guard conditions* that must hold for the edge to be enabled. The behaviour of the machine is the following: the current state contains the set of actions executed at the moment, which is performed until one of the events associated to the outgoing edges happens. Whenever such an event is sensed by the agent, we say that the event *triggers* the transition which makes it available for execution. For the edge to be actually enabled at that time another condition must be met, namely the guard of the transition must hold. When an edge is *triggered* (the associated event happens) and *enabled* (its guard condition holds) it is allowed to be followed and the next state represents the set of actions the agent is to execute next. If an action was present in the previous plan state and it is not in the next one, that action must be terminated. On the other hand, if an action appears in the next state it must be started. All actions that are both in the previous and next plan state keep being executed. To make the operational semantic clearer we assume that all events are external (i.e. they cannot be generated by the machine itself) and transitions are instantaneous, so that no event can be lost during a transition execution. If the machine is hierarchical, the edges are processed from the top level to the bottom one, and if an edge at a higher level is triggered and enabled, that preempts any edge at

lower levels. Final states are absorbing states that cannot be left once entered and determine the execution termination.

In our example, the leftmost state has two outgoing edges with a guard condition each. Every time the machine is in that state both the edges are triggered (if no event is explicitly specified the empty event is assumed, which is always triggered) and since the conditions, [BallSeen] and [not BallSeen], are mutually exclusive, exactly one must be enabled and will be followed. Such a structure simply represents an IF-statement common in any programming language. In the same way as events, when no condition is specified the *true* condition is assumed, which is the condition that always holds. Two, or more, edges triggered by the same event, and guarded by non mutually exclusive transitions, might be triggered and enabled at the same time. Such a situation characterises a non deterministic choice that can be interpreted as a choice between different plans. This way, a transition system represents a set of plans, that divert at the choice points and may or may not join later on.

3.1.2 Plan schemas

After having informally introduced the representation, we are going to provide a formal definition of the learning framework. We refer to our transition systems as *plan schemas*, which can be considered either as a set of plans merged at the points they share, or as a partially specified plan in which non-determinism characterises a choice left unspecified:

Definition 4 (Plan Schema)**.** *A Plan Schema is a tuple* $\langle S,\ s_0,\ F,\ E,\ \Phi,\ A,\ L,\ T\rangle$ *where:*

- S *is a finite set of plan states*
- s_0 *is the initial plan state*
- $F \subset S$ *is a set of final plan states*
- E *is a finite set of events*
- Φ *is a set of conditions*
- A *is a set of actions*
- $L : S \rightarrow \wp(A)$ *is a total labelling function that maps plan states on actions*
- $T : S \times E \times \Phi \rightarrow S$ *is a transition relation augmented with a trigger and a condition. For each* $s \in S$, $e \in E$ *and* $\phi \in \Phi$, ϕ *must entail the pre-conditions of all the actions in* $L(T(s, e, \phi))$

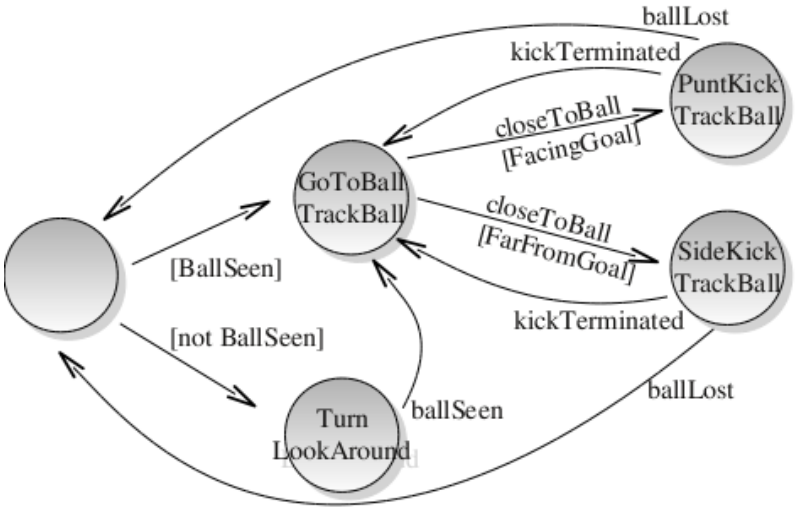


Figure 3.2: A non-deterministic choice point in the domain of RoboCup Soccer

If the state machine is deterministic (it can never happen that two transitions are triggered and enabled at the same time), then the plan schema is actually a single plan since no choices are left to the executor and the entire behaviour is specified. On the other hand, if the machine is non-deterministic, the plan schema represents multiple plans and each non-deterministic choice is a fork among them. Nothing prevents different plans from sharing common paths and depart only where their behaviour differs.

Consider again the example of the football playing robot, and let *PuntKick* and *SideKick* be two different kick actions. `PuntKick` needs to be executed when the agent is facing the goal, while `SideKick` is meant for when the agent is far from the goal. The two actions are made available to the agent refining the plan of Figure 3.1 as shown in Figure 3.2. The event `closeToBall` triggers the termination of the set of actions {`GoToBall`,`TrackBall`} during the execution of which the agent approaches the ball while keeping it on sight. At this point the two kicks are available, each with its precondition, that is expressed in the edges' guards. The two conditions are not mutually exclusive, so it may happen that when `closeToBall` is perceived the agent is both facing the goal and far from it. In this case two plans are available, namely the plan that when the agent is facing the goal and far from it executes a `PuntKick`, and the plan that executes a `SideKick`. In such non-deterministic choice points we intend to make an informed choice, and *learn* which behaviour performs best in practise, depending on the

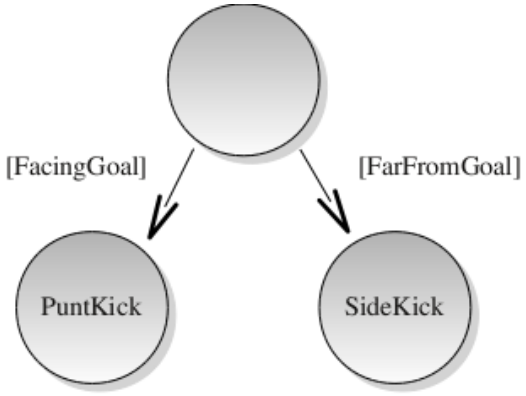


Figure 3.3: A simple example of a sub-procedure

adversaries, or the many aspects that the planner couldn't take into account.

The states in which {PuntKick, TrackBall} and {SideKick, TrackBall} are executed share the same set of outgoing edges, therefore are good candidates to be factored out and separated in a sub-procedure. The action `Kick` of the plan schema in Figure 3.1 can indeed be the machine represented in Figure 3.3. Both the actions `PuntKick` and `SideKick` generate the event `kickTerminated`, and if during any of them an event `ballLost` should be detected, the procedure would be terminated, regardless of the action currently in execution.

3.1.3 On events and conditions

Events and conditions are not sharply separated concepts; it is, to some extent, up to the designer to decide what will be represented by which.

The main difference between the two categories lies in timing. Events are pinned to a timestamp, they are detected in a precise moment, are instantaneous (their perception is assumed instantaneous, in practise it will take time, but the time of the computation should be irrelevant compared to the rate at which the environment changes) and we assume that two events cannot be perceived at exactly the same time. Conditions, on the other hand, represent a partial description of the environment and as such describe regions of the environment state space. When the KB entails a condition the agent *believes* to be in that region, and two conditions can be true at the same time, if their respective regions intersect. A condition may hold for any amount of time.

Some particular regions of the environment state space might have both a condition and an event associated to them: the condition to characterise the re-

gion, the event to mark the moment in which such a region is entered. So, for instance, plan schemas can represent sets of plans of classical planning, where actions are operators characterised by pre- and post-conditions, and the goal is also described by a formula. In such a setting, every plan state has a single action; the pre-condition of each action maps to the guard of the edge towards the state with that action; the post condition maps to an event corresponding to the condition becoming true; finally the goal maps to an event that takes the machine to a final state where no action is executed.

Events, moreover, can represent aspects that can be particularly useful for reactive plans, especially in robotics, and are quite cumbersome if taken into account by adding them to the description of the environment state space. For instance, if a timeout that expires, or a message received, or a joint that has reached its target position, should be represented by creating a binary variable in the state representation, that would duplicate the space unnaturally. Events do not impact the size of the state space, and mark a distinction between the aspects of the environment that call for a new decision and those upon which such a decision must be made.

3.2 Learning Framework

The learning framework is focused on exploiting the non determinism of a plan schema to make an informed choice.

Reinforcement Learning allows us to make use of experience to improve an agent's performance over time and seems a reasonable choice to achieve our goal. RL has been thoroughly studied within the MDP framework, since this framework provides a formal and neat mathematical notation for analysing an important class of sequential decision problems. In traditional RL applications it is assumed that all the relevant knowledge about an agent's environment can be encoded in a structure, usually a Markov Decision Process (MDP). Moreover, both in "model-free" and "model-based" RL techniques, it is assumed that even though the agent might not know exactly what the structure of the MDP is (e.g., the transition matrix), all sample observations are drawn from some underlying MDP. In the class of problems we are considering, however, assuming the existence of a fully observable MDP is usually unrealistic. Even trying to come up with a reasonable possible encoding for the states, which could somehow guarantee that the Markovian assumption is respected, might be infeasible.

For this reason we rely on a generic knowledge base that reflects the beliefs of the agent about the environment, without building a dynamic model of it. In the following, we will define a stochastic decision process by deriving it from the plan and we will use this model to gather the experience to use in subsequent trials.

The state of the system is composed by both the state of the plan and the state of the environment, but the latter is in general not completely known. The reward depends on how the state of the environment is perceived by the agent. In order to make a decision in non-deterministic choice points, we want to look forward in the plan having a value function associated with plan states, but not looking forward in the environment state space trying to predict the outcome of actions (i.e., the next environment state).

The plan executor must adhere to the state machine operational semantic as long as the choices are deterministic. Whenever a non-deterministic choice must be taken, the executor can refer to the value function to evaluate the alternatives and then exploit or explore as usual in RL.

3.2.1 Definition of a controllable stochastic process

In order to properly characterise the stochastic process associated to the previously described state machine, and to set the proposed method in the RL framework, we define it in terms of a Timed Non-Markovian Decision Process (TNMDP). We call it so, since it borrows from SMDPs (cf. Section 1.3) the way actions with different durations accumulate the reward, and is Non-Markovian because, in general, the description of the environment given by conditions and events will not be enough to predict the reward from each state. This latter aspect will be explained more in detail later.

We first show the construction of the TNMDP with the example of the previous chapter, and then provide its formal definition. Let's consider the non-deterministic choice point shown in Figure 3.2, extrapolated and reported for ease of reference in Figure 3.4

The nodes that allow for non-determinism (i.e., that have more than a transition associated with the same event, and whose guard conditions are not mutually exclusive) are split into a number of nodes equal to the constituent events of the conditions. In the example, the event *closeToBall* is associated to the conditions `FacingGoal` and `FarFromGoal`. This gives four possible constituents, namely: only `FacingGoal` is true, only `FarFromGoal` is true, both are true or none of them is.

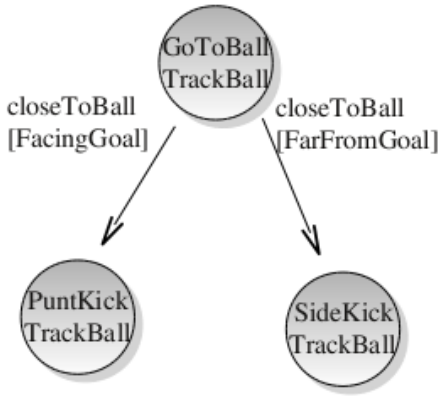
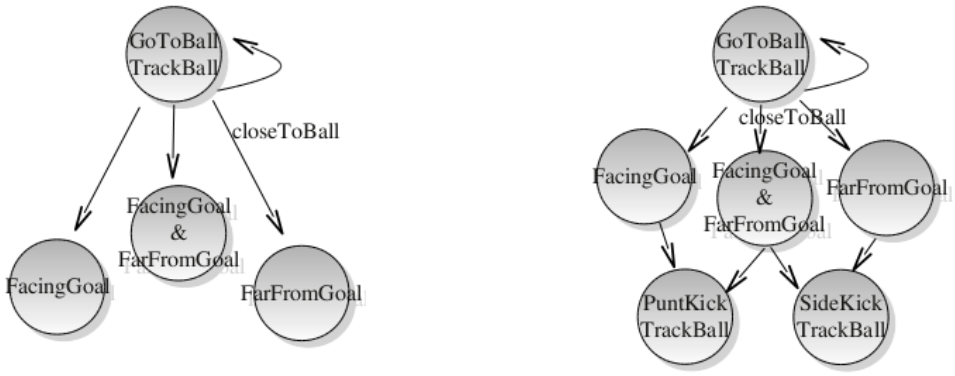


Figure 3.4: The non deterministic choice point from the example of Figure 3.2



(a) First step

(b) Second Step

Figure 3.5: The transformation of the non deterministic choice point into an TNMDP

To the first three we associate a state and an arc from the initial state, in which the actions `GoToBall` and `TrackBall` are executed. The last situation, in which none of the conditions holds, translates into a loop on the initial state. The result of the first step of this transformation is represented in Figure 3.5 (a).

In this section we make use of the term "action" as it is done in the literature of stochastic processes when we refer to the TNMDP. Therefore, while an action in the plan schema is the actual intervention of the agent in the environment, an *action* in the TNMDP is an instantaneous transition available to the controller. An *action* in the TNMDP causes a change in the state of the process, but cannot modify the state of the environment while this is the primary intention of an

action in the plan schema. All the created edges correspond to the same non-deterministic action of the TNMDP reported as *closeToBall*. Since it is caused by the perception of the event *closeToBall*, the result of this action depends on the environment and cannot be chosen by the agent.

Next, each node associated to a constituent of the conditions is connected to the action node containing the actions enabled by that constituent. In our example, *FacingGoal* is connected to the node representing the action *PuntKick*, while *FarFromGoal* is connected to *SideKick* and *FacingGoal&FarFromGoal* is connected to both. At this level the edges reaching different action nodes are associated to different *actions* of the TNMDP. The resulting graph has a choice point in the state *FacingGoal&FarFromGoal* since in that case two actions are simultaneously available.

The number of nodes in which a choice point in the original plan is split is exponential in the number of conditions. This is not surprising, as in the case of full observability and discrete state space this number would be equal to the number of states storing an entire Q-function. Nonetheless, the underlying assumption is that the domain is continuous and partially observable so that there is no notion of single state, and considering single states or many small regions is not possible nor desirable. Hence, even if it is possible to consider function approximation, we are not going to need it necessarily.

To give a formal definition of the TNMDP we have informally previously introduced, we define the set $C_{cnd}(s,e)$ of the constituent events generated by overlapping conditions in a specific choice point (denoted as $\langle s,e\rangle$) of a plan schema $PS = \langle S, s_0, F, E, \Phi, A, L, T, \rangle$ as follows:

$$C_{cnd}(s,e)\begin{cases} = & \wp(\{\phi_k\}) \setminus \emptyset \text{ if there exist } k \text{ conditions} \\ & \phi_1 \dots \phi_k \text{ and a state } s_j \text{ s.t.} \\ & \langle s, e, \phi_i, s_j\rangle \in T \; \forall i \in \{1,\dots,k\} \\ = & \emptyset \text{ otherwise} \end{cases}$$

In our example

$$C_{cnd}(\{GoToBall, TrackBall\}, closeToBall) = \{ \begin{array}{l} \{FacingGoal\}, \\ \{FarFromGoal\}, \\ \{FacingGoal, FarFromGoal\}\} \end{array}$$

Next, we define the set S_c of the states generated by condition overlapping in all choice points:

$$S_c = \{\langle s, e, cond\rangle | s \in S, e \in E, cond \in C_{cnd}(s,e)\}$$

In our example

$$\begin{aligned} S_c = \{ \quad & \langle \{GoToBall, TrackBall\}, closeToBall, \{FacingGoal\} \rangle, \\ & \langle \{GoToBall, TrackBall\}, closeToBall, \{FarFromGoal\} \rangle, \\ & \langle \{GoToBall, TrackBall\}, closeToBall, \{FacingGoal, FarFromGoal\} \rangle, \\ & \langle \{PuntKick, TrackBall\}, kickTerminated, \{true\} \rangle, \ \ldots \ , \ \} \end{aligned}$$

Where the last state has again been taken from the full plan of Figure 3.2, to illustrate the case of an event with no condition associated. Those states constitute the second layer of Figure 3.5 (b). Finally, we also define a function S_c^e to select in S_c the states that are generated by a specific choice point as follows:

$$S_c^e(s, e) = \{\langle s, e, cond \rangle \in S_c\}$$

and a function C_{cnd}^s to extract from the states in S_c the set of conditions:

$$C_{cnd}^s(\langle s, e, cond \rangle \in S_c) = cond$$

Time has not been addressed yet. We consider time in discrete time steps and actions can take multiple time steps to complete. We use the following notation (Bertsekas, 1995):

- t_k: the time of occurrence of the k^{th} transition. By convention we denote $t_0 = 0$
- $s_k = s(t_k)$ where $s(t) = s_k$ for $t_k \leq t < t_{k+1}$
- $a_k = a(t_k)$ where $a(t) = a_k$ for $t_k \leq t < t_{k+1}$

We define a Timed Non-Markovian Decision Process $TNMDP = \langle \ S_{sp}, \ A_{sp}, \ T_{sp}, \ \rho_{sp} \rangle$ such that:

- $S_{sp} = S_c \cup S$, is the state set. S_c is the set generated by overlapping conditions, whereas S is borrowed directly from the plan schema and accounts for *action states*, that is states that are not the result of a choice point split but are associated to actions in execution. The first and last layer of Figure 3.5 (b) are an example of the states in S while the intermediate layer is an example of the states in S_c.
- $A_{sp} = \{a \in \wp(A) | \exists s \in S \ s.t. \ L(s) = a\} \cup E$, is the action set. The first part is the co-domain of the labelling function in the plan schema. We create an action for each possible set that labels the states of the plan schema. Notice that those actions are deterministic and we give them the same name as their target state. In our example of Figure 3.4 the co-domain of labelling function is

{{GoToBall,TrackBall}, {PuntKick,TrackBall}, {SideKick,TrackBall}}. You can spot the corresponding actions in Figure 3.5. The set E (events in the plan schema) is used to define the actions on which the agent has no control. These actions are non-deterministic and their outcome depends on the environment. Again, in Figure 3.5, *closeToBall* is an example of such an action.

- $T_{sp}(s, a, s', \tau) = Pr(t_{k+1} - t_k \leq \tau, s_{k+1} = s' | s_k = s, a_k = a)$ is the probability for action a to take τ time steps to complete, and to reach state s' from state s

  – if $a \notin E$: the action is deterministic. An action that *is not in* E connects a state in S_c to the state in S (second to third layer in the example) labelled with the actions enabled by the condition in that state. Moreover, those actions do not reflect any change in the environment so they always complete in zero time. That is,

$$T_{sp}(s, a, s', \tau) \begin{cases} = 1 \text{ if } & \exists\, s_i, e, \phi. \\ & \langle s_i, e, \phi, s' \rangle \in T \wedge\ s \in S_c^e(s_i, e) \\ & \wedge\, \phi \in C_{cnd}^s(s) \wedge\ L(s') = a \wedge\ \tau = 0 \\ = 0 & \text{otherwise} \end{cases}$$

  A state s is connected to the state s' by a iff s is a state generated by a condition constituent, it is linked to s' by the plan schema, and a is the label of that link.

  – if $a \in E$: the action is non-deterministic. These actions take the time spent in the previous state waiting for the event. An action that *is in* E connects a state in S to itself and to all the condition states that its split generates (first to second layer in the example). Therefore, events cannot connect all pairs of states, which translates into:

$$T_{sp}(s, a, s', \tau) \begin{cases} = 0 \quad \text{if } s \notin S\ \vee s' \notin S_c^e(s, a) \cup \{s\} \\ = \int_H p(t_{k+1} - t_k = \tau, s_{k+1} = s' \\ \qquad |s_k = s, a_k = a, \vec{h}) p(\vec{h})\, d\vec{h} \\ \text{otherwise} \end{cases}$$

  If a connection between s and s' through e exists according to the plan schema, the value of the transition function is the probability for the event a to happen in the state $\langle s, \vec{h} \rangle \in S \times H$ where H is the domain of (continuous) hidden variables. Since those variables are not observable,

the sample distribution is the (hidden) underlying one marginalised over the hidden variables. This makes the stochastic process non Markovian due to partial observability.

- $\rho_{sp}(s, a, s', k, r) = Pr(r_{t+k} = r | s_t = s, s_{t+k} = s', a_t = a)$ is the reward function. As for MDP and SMPDs (cf. Section 1.1 and 1.3) we denote with $\rho_{sp}(s, a, s', k) \sim \rho_{sp}(s, a, s', k, .)$ the reward extracted from ρ. We define its value to be 0 if $a \notin E$. Therefore the immediate reward is non-zero only for events (which can take time).

In order to define a decision problem, we establish a performance criterion that the controller of the stochastic decision process tries to maximise. As such, we consider the expected discounted cumulative reward, defined for a stochastic policy $\pi(s, a)$ and for all $s \in S_{sp}$ and $a \in A_{sp}$ as:

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}^\pi \left[\sum_{i=1}^{\infty} \gamma^{i-1} r_i \middle| s_0 = s, a_0 = a \right] \\ &= \sum_{s' \in S_{sp}} \sum_{\tau=0}^{\infty} T(s,a,s',\tau) \Big(\sum_{t=0}^{\tau} \gamma^t \rho(s,a,s',t+1) \; + \; \gamma^{\tau+1} V^\pi(s') \Big) \end{aligned} \quad (3.1)$$

The optimal discounted reward function is defined as:

$$Q^*(s,a) = max_\pi \, Q_\pi(s,a), \; s \in S_{sp}, \; a \in A_{sp} \quad (3.2)$$

3.2.2 Markovian and non-Markovian rewards

In order to understand why the stochastic process resulting from the plan is in general non-Markovian, consider the following example again from the RoboCup Soccer domain.

A robot is positioned at the edge of the penalty area and is about to shoot a penalty kick. It starts facing the goal with the ball in front of it. Kicking straight results in a goal and a reward of 10 with probability 1, while turning before kicking causes the robot to miss the ball and get a reward of zero after a timeout has expired with no goal scored. The robot executes the plan shown in Figure 3.6. Since the plan is very simple, there are no conditions and not more than an event connected to each state, the stochastic process generated is not significantly different from the plan itself. So, for the sake of the argument, let's just consider this plan as a stochastic process, in which the only choice left to the controller is in the initial (leftmost) state. When the transition with the event `goal` is executed

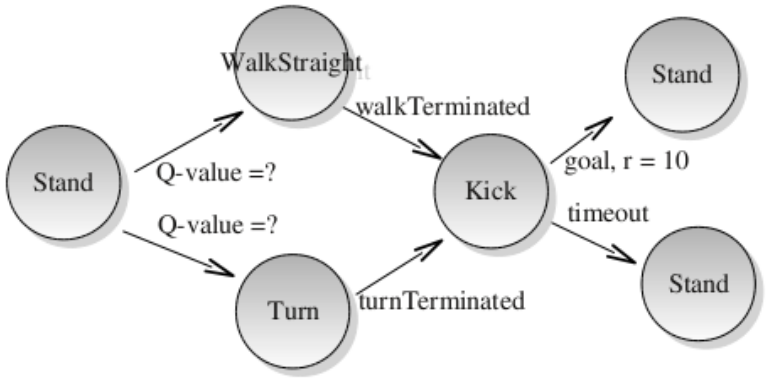


Figure 3.6: Example of a non Markovian reward

the agent receives the reward of 10, while all other transitions give a reward of 0. Finally let's assume that the task is restarted indefinitely so that each state and each action can be visited infinitely often.

When the agent executes `Kick` after `WalkStraight` it faces the goal and scores every time, while when it kicks after having executed `Turn` it faces the wrong direction and never scores. Q-learning would give both the edges that enter `WalkStraight` and `Turn` the same value, since they lead to the same state (`Kick`) with no intermediate reward. Their effects, though, are clearly different. In this example all transitions are deterministic, therefore Markovian. What is not Markovian is the reward in the state `Kick` that depends on the robot's orientation which is not taken into account. This particular example is an hPOMDP, that is one of those partially observable processes in which the history is enough to discriminate the actual state. Rewriting the plan as in Figure 3.7 it would be possible to learn the correct behaviour. This is not possible in general, however, and even when it is possible it duplicates the states creating a tree that grows exponentially with the number of choice points.

3.3 Summary

We have introduced a translation of a partially specified plan into a controllable stochastic process. The framework allows for both constraining the search in the

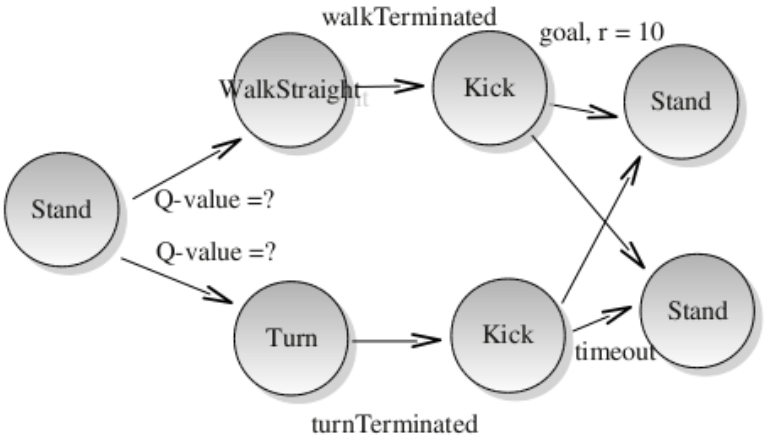


Figure 3.7: A different structure for the formed example that allows to distinguish the actual state

space of the possible policies, and loosen the requirements on the accuracy of the model used for planning. Plans can be hierarchical, calling sub-plans instead of atomic actions. From the topmost plan down, the executor is triggered by events and verifies conditions on a knowledge base. When a sub-procedure is called in a state, the execution of the calling plan is suspended and the reward accumulated for that state. Each layer of the hierarchy learns separately, aiming at *recursive* rather than *hierarchical* optimality (Dietterich, 2000). Moreover, each layer faces a learning problem that is in general non-Markovian, as explained in Section 3.2.2. For this reason, Chapter 4 is devoted to the definition of an algorithm for learning in such a context.

4

Learning Algorithms

After having introduced a framework for Hierarchical Reinforcement Learning allowing partial knowledge and parallel actions, we analyse the problem of learning a policy in such a challenging context. There are a few domains in the literature in which, despite the fact that the observations make it difficult to distinguish different states, TD(λ) can learn the optimal policy. The *eligibility trace*, that is the multi-step, exponentially decaying, trace along which the updates are performed, can help sort out the differences where the previous history matters. When the reward is affected by factors external to the agent though, credit assignment is particularly complicated. In the following, we first try to highlight the respects in which an algorithm for policy iteration based on TD(λ), such as Sarsa(λ), fails at estimating the value function. Then, we try to overcome the limitations with a new algorithm that is created on the basis of considerations about real world domains, the theory of chapter 1.5, and stochastic search.

4.1 What is wrong with *direct* RL

It has already been shown (cf. Section 1.5.3) that the policy followed by the agent while learning affects not only the value function learnt (as one would expect from any on-policy method), but also the capability to converge to the value function

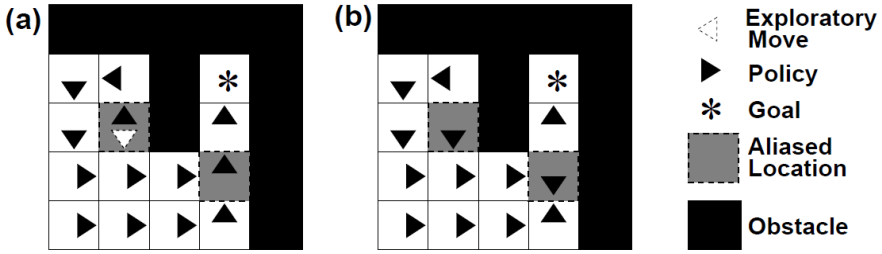


Figure 4.1: An example on consistent exploration

of the current policy. Q-learning and Sarsa can cycle through different policies, although the overall performance is not necessarily worse than a stable but sub-optimal policy. In the next sections we consider two sources of difficulties for direct methods: exploration and estimation.

4.1.1 Consistency

It is common in MDPs that every time the agent encounters a state a new decision is made. When the method used to learn the value function is multi-step though, if a state is encountered multiple times during the same episode, the choice made at later times can affect the value of the choice made the first time. This causes the value of the two choices to combine, making it hard to distinguish which one caused the reward. Crook (2007) proposes the following example to underline the importance of a *consistent* exploration. Consider the grid world in Figure 4.1 in which grey states are aliased, that is, indistinguishable. If the agent explores moving south in the left-most aliased state, and exploits the current value function going north from the right-most one, it reaches the goal faster than if it went north from the first state. The higher value for the action `south` in the first state is obtained through a sequence of actions that can be performed only by a stochastic policy, and we indeed know that stochastic policies on NMDPs can perform arbitrarily better than deterministic policies (cf. Section 1.5.1). As long as we are looking for a deterministic policy, however, this kind of exploration, that performs actions that a deterministic policy may not do, can be misleading. Therefore, the first element of our algorithm is a *consistent* exploration, that was already present in MCESP in the definition of the value function of equation 1.10,

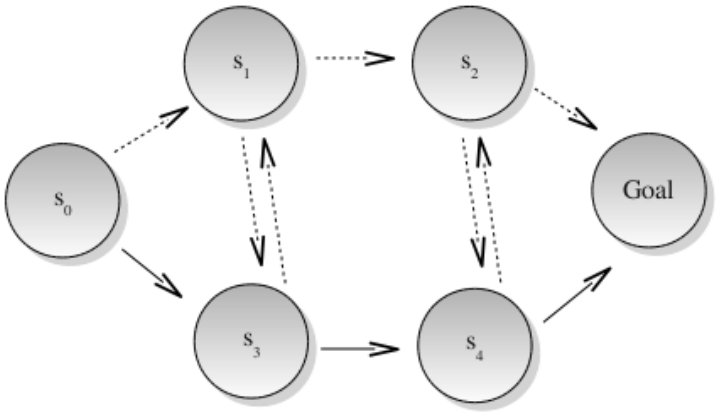


Figure 4.2: An NMDP in which the exploration can damage the current estimation of the value function

reported here for ease of reference:

$$Q^{\pi}_{o,a} = \mathbb{E}^{\pi \leftarrow \langle o,a \rangle} \left[R_{post-o}(\omega) \right]$$

In this equation $\pi \leftarrow \langle o, a \rangle$ is followed, instead of just executing a in o once, which means that every time the observation o is perceived, the action a must be executed. In our algorithm though, we implement a global search method, allowing more than one *exploratory* step per episode. Our implementation makes a decision for each observation the first time that it is encountered, and then remembers it in case that observation should be seen again.

4.1.2 Action values

In non-Markovian domains, an exploratory move in some part of the state space can ruin the estimation of the value of another action in some other part of the space. Consider, for instance, the NMDP in Figure 4.2. The actions with solid lines form the optimal path from s_0 to the goal state. Any dotted action is on a path that gets a reward much lower than the optimal one. If the agent follows the route $\langle s_0, s_1, s_2, s_4, goal \rangle$ it receives a low reward and updates its estimate of all the actions on the way. The last, optimal, action is incorrectly underestimated, and its value can become lower than the value of its alternative action, possibly causing it to be wrongly not chosen again. If it is not chosen along the optimal path, that is the agent reaches s_4 and goes to s_2, the value of the whole path is underestimated, and the value function completely disrupted.

In an MDP, the expected cumulative discounted reward can be temporarily underestimated too, if *after* an action is executed, some sub-optimal action is taken. In that case though, the correct value will be eventually restored, since the value depends only on the current state. In MDPs, as the action-values closer to the reward get more precise, the right estimation is propagated backward. Therefore, if the agent acts greedily almost always, even an on-policy method can converge to the right value.

In order to prevent the behaviour just described, we introduce two elements in our algorithm. First, if at least one exploratory move is taken, the whole episode is marked as part of the exploration and treated accordingly. Second, after an exploratory episode, the value function is updated only on those state-action pairs that have received a reward higher than the current one. This means that an exploratory episode cannot lower the value of any action. This way, a state-action pair that is not part of the optimal policy will keep raising under any policy that returns a value higher than the optimal one. Every action not currently optimal gets overestimated (since its value cannot be lowered, it converges to the maximum ever seen) and is not taken into account as long as its overestimation is lower than the average of the current best action for that state. In the following section the algorithm is described in detail.

4.2 The algorithm: SoSMC

As mentioned in the background section, if we do not rely on the Markov property, there must be some other aspect of the domains that makes our algorithm more effective than just random searching. We informally introduce here the ideas behind the algorithm, motivating some of them with assumptions on the domains:

- **Stochastic search**. Since we assume that the NMDP is finite, the number of possible policies is also finite. By making sure the probability to select each policy is non-zero, we guarantee that every policy is, in the limit, sampled infinitely often. Instead of just randomly picking policies, however, we bias the search towards the ones that look more *promising*, in a sense that will be explained later.

- **Value function**. Several methods that search directly in policy space usually represent the policy explicitly (e.g., policy gradient (Sutton et al, 2000)), rather than computing the value function. We try to use the value function

in a different way: for the policy currently in use, the value of the actions converge to the expected cumulative discounted reward as usual, while for any other action, the algorithm stores an *upper bound* on the value of that action, given the knowledge acquired until then. Confidence bounds have been used on MDPs (Auer, 2002; Auer et al, 2009) building a model of the system and estimating uncertainty. In our algorithm, we cannot tighten the bound along the way, but we try to keep it updated, that is to memorise the highest value seen to compare it to the current average value, and perform a sort of branch and bound on action values.

- **Locality**. The value function is updated globally, that is, if a policy performed better than the current one, all the action-values that have been found higher than the expectation are updated simultaneously. This is in contrast with MCESP that implements a local search, therefore updating one action-value at a time. On the other hand, decisions are made locally, state by state, and possibly independently in different parts of the hierarchy (as opposed to ALisp, cf. Section 1.4.3).

- **Stability**. Action-values in NMDPs depend not only on the state but also on the policies under which they have been computed. For such values to make sense, the policies sampled must not be totally unrelated and far apart in the policy space. This is the main motivation behind MCESP's local search: no more than one choice at a time can be reliably evaluated. We admit the possibility that an action-value is the result of blended policies, if those policies are not too far, but still farther than the one-step distance allowed by MCESP. What we mean by *related* policies will be clarified in the next section.

- **Regularities**. We know that there exist worst cases in which there is nothing better to do than evaluate every policy. Unfortunately, at the rate at which the number of policies grows with the number of states, performing such a search in any reasonable amount of time is hopeless. We have to rule out, then, all those problems in which the solution can only be found by combinatorial optimisation: no needles in haystacks. Reality usually proves to be less pathological than worst cases. We informally assume that it is unlikely (not impossible, but at least improbable) that a policy with a low mean can give extremely high rewards. Moreover, although we know that actions in different parts of a sequence may be related to each other, we assume that

this does not affect the whole policy. Therefore, there are parts of the policy that are independent of others, and the choices on those localities can be made independently.

The main idea behind the algorithm, Stochastic Search Monte Carlo (SoSMC), is based on the intuition that often a few bad choices disrupt the value of all the policies that include them. Taking those policies as if they were as valuable as any other just wastes samples. We would rather like to realize that those actions are not promising and not consider them unless we have tried all the other options. The strategy would consider all the policies with a probability proportional to how *promising* they are, which we believe is beneficial in at least two ways: (1) the algorithm reaches the optimal policy earlier; (2) during the phase of evaluation of those *promising* but suboptimal policies, the behaviour is as good as the current information allows.

The algorithm (cf. Algorithm 1) is constituted by two parts: the *exploratory* phase and the *assessing* phase, as described in the next section.

4.2.1 Exploration: gathering information

The exploration initialises the Q-function to drive the execution in the subsequent phase. The aim of the initial exploration is to determine an upper bound for each state-action pair. For a number of episodes exp_length the agent chooses a policy according to some strategy Σ (e.g., uniformly at random), and in each pair $\langle s, a\rangle$ stores the highest value that any policy, going through $\langle s, a\rangle$, has ever obtained. Consider the simple example of the N-MDP in Figure 4.3(a). This N-MDP has three states and four actions with a total of four policies. Let the reward returned by each of those policies be normally distributed, with means and standard deviations represented in Figure 4.3(b). Figure 4.3(c) and 4.3(d) show the value of the Q-function for each action during a particular run. The first 100 episodes belong to the exploratory phase, in which the actions A1 and A2 obtain the highest reward, making the policy A1-A2 look particularly promising. An action is considered as *promising* as the highest value of the reward that choosing that action has ever given. In the case of A1-A2, its good result is due to the high variance, rather than the highest mean. This aspect will be addressed by the second phase of the algorithm.

The number of episodes in the exploratory phase should ideally allow for the sampling of each policy above its mean at least once. Depending on the

Algorithm 1 SoSMC

$exp\_length \leftarrow$ number of episodes in the exploratory phase
$n \leftarrow$ current episode
$t \leftarrow$ last exploratory episode
$\alpha(n, o, a) \leftarrow$ learning step parameter
initialize $Q(s, a)$ pessimistically
{Exploratory phase}
for $i = 1$ to exp_length **do**
  generate a trajectory ω according to a policy π extracted from a strategy Σ
  for all $o \in O$, $a \in A$ $s.t.\langle o, a\rangle$ is in ω **do**
    $Q(o, a) = max(Q(o, a), R_{post-o}(\omega))$
  end for
end for
{Assessing phase}
for all other episodes : n **do**
  if n is such that the current estimate is considered accurate **then**
    $t = n$
    $\pi \leftarrow$ a policy chosen from Σ
  else
    $\pi \leftarrow$ the last policy chosen
  end if
  {Possible policy change after an exploratory episode}
  if $n = t + 1$ **then**
    $\pi \leftarrow$ the policy that greedily maximizes Q
  end if
  generate a trajectory ω from π
  if $n = t$ **then**
    for all $o \in O$, $a \in A$ $s.t.\langle o, a\rangle$ is in ω **do**
      $Q(o, a) = max(Q(o, a), R_{post-o}(\omega))$
    end for
  else
    for all $o \in O$, $a \in A$ $s.t.\langle o, a\rangle$ is in ω **do**
      $Q(o, a) = (1 - \alpha(n, o, a))Q(o, a) + \alpha(n, o, a)R_{post-o}(\omega)$
    end for
  end if
end for

particular strategy Σ and the shape of the distributions of the policies, such a number for exp_length might be computable. In practice, unless the problem is effectively hierarchically broken into much smaller problems, the number of episodes required is hardly feasible. In those cases, the exploration has to be shorter than what would be required to complete, and the algorithm will start with

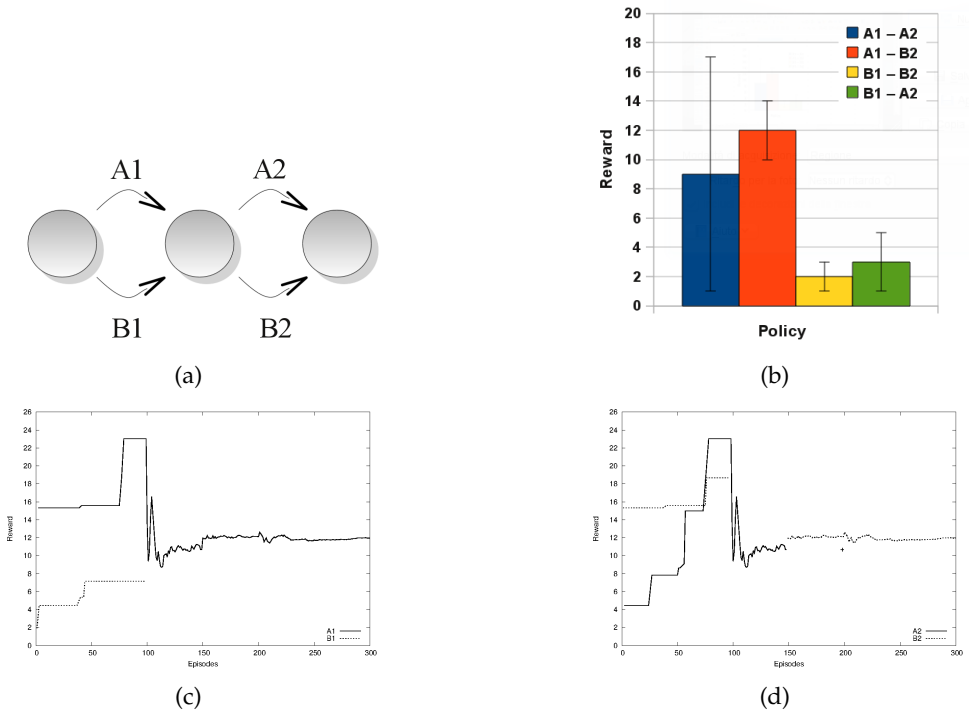


Figure 4.3: A simple example of an NMDP (a). The four policies return a reward normally distributed whose means and standard deviations are shown in (b). The evolution of the Q-function for the first state (actions A1 and B1) is represented in Figure (c), while for the second state (actions A2 and B2) is represented in Figure (d).

an upper bound for the limited number of policies visited, and keep exploring during the second phase.

If the domain does not allow for the estimation of a helpful upper bound, for instance because every action can potentially give high rewards, the first phase can be skipped initializing all actions optimistically. We conjecture that this may happen on synthetic domains in which the stochasticity is artificially injected, but it is rarer in real-world applications.

4.2.2 Assessment

We want to maximize the *expected* cumulative discounted reward, rather than the maximum obtainable one, therefore an evaluation of the *promising* policies is needed.

A random searching algorithm is an algorithm that picks a candidate π_k, at

iteration k, from a given distribution on the space of all possible policies. It evaluates π_k and either updates the current best solution or discards the candidate. We rely on the main theorem behind stochastic search (Spall, 2003, pag. 40) which states that if the search space if finite and each point has a non-zero probability to be picked, then $\pi_k \to \pi^*$ almost surely as $k \to \infty$. In order to guarantee that the conditions expressed by the theorem are met we: (1) limit the search space to the *finite* set of deterministic stationary policies; (2) require immediate rewards to be bound; (3) require that the strategy Σ according to which the policies are picked assigns a non-zero probability to each of them. Since the value of optimal solutions is, in general, not known in advance, there is no obvious stopping criterion. In practice, and as it is common in stochastic search, we may define a threshold above which we accept any solution.

In the second phase the algorithm picks a policy according to Σ, evaluates it with first-visit Monte Carlo, and stops if the policy's reward is above a threshold. Monte Carlo methods wait until the end of the episode to update the action-value function, therefore the task needs to be episodic. The novel aspect of SoSMC is the way in which the search is biased. Reinforcement learning algorithms can traditionally be considered as composed by prediction and control. While the prediction part is borrowed from the literature (first-visit MC, and Perkins's definition) the control part is based on the estimate of upper bounds, their storage in the action-value function, and their use to generate the next policy to try. Moreover, differently from MCESP, it performs a global search. It also employs a *consistent* exploration Crook (2006), that is, during the same episode, every time the agent perceives an observation it performs the same action.

If the reward is deterministic a single evaluation per policy is sufficient. Such a case may, for instance, occur on POMDPs in which the underlying MDP is deterministic and there is a single initial state. If the reward is stochastic, on the other hand, the capability to have an accurate estimate of the reward depends on the distribution. For some distributions it may be possible to compute confidence bounds and ensure that the reward returned by a policy is higher than the threshold with some probability. In general, the estimated mean cannot be guaranteed to be correct after any finite number of samples. In such cases, we use a fixed number of samples k which empirically proves to be reliable. While losing some of the theoretical guarantees, we experimentally show how SoSMC can outperform the best results in the literature for different domains. While an inaccurate estimation of a policy may deceive the stopping criterion, if the thresh-

old is not too tight on the optimal value a *good* policy is in practice always found in the domains we have used. The example of Figure 4.3 shows an assessment phase with $k = 50$.

4.3 Choosing the parameters

Different choices are possible for the exploration strategy Σ and for the schedule of the step-size parameters α, actually making SoSMC a family of algorithms.

For what concerns the exploration strategy, we have used both ϵ-greedy and SoftMax (cf. Section 1.5.3). In the case of ϵ-greedy (where we refer to the algorithm as ϵ-SoSMC), the choice has been made for each state the first time it is encountered, and then remembered throughout the episode. Therefore, the policies closest to the current optimal one are more likely to be selected, and become less and less probable as the distance from the optimal policy increases. This is meant to implement that principle of *stability* mentioned in Section 4.2, that is to avoid the sampled policies to leap irregularly through the policy space, since the value function would not be able to track them. As for SoftMax, again the current optimal policy is the most likely to be selected, but the neighbourhood is considered not just in the distance from such a policy, but also in the value of its actions, evaluated locally. SoSMC with SoftMax, referred to as Soft-SoSMC, performs particularly well in those domains in which the combinatorial aspect is minimal, and the choices can often be made separately. If some actions have low values, regardless of the choices made elsewhere, SoftMax can avoid selecting those actions making a better use of the samples. In particular for the first, exploratory phase, SoftMax with a decreasing temperature, as the exploration becomes closer to the end, allows to focus on that set of policies that has given the highest rewards and proved to be more effective than ϵ-greedy.

For what concerns the step-size parameter, we identify at least two possible schedules. If α is set to a constant lower than $1/n(s, a)$ (where $n(s, a)$ is the number of episodes in which the state-action pair $\langle s, a \rangle$ has been visited since the last policy change) the value of an action is affected by different policies. Depending on the domain this might or might not be helpful on the probability for that action to be picked again. We empirically found that for the actions that are better or worse than their alternatives regardless of the rest of the policy, an estimate that is built across policies has a favourable impact on the selecting distribution. If $\alpha = 1/n(s, a)$ on the other hand, the estimate is memoryless and

every policy change produces a new one for each state-action pair visited. This is the case of the example in Figure 4.3

4.4 Credit assignment for parallel execution

When the agent can perform more than one action at the same time, credit assignment is complicated by the fact that it is hard to distinguish whether it was the particular combination of actions that generated the reward, or some of the actions alone. Determining what should be rewarded is then non-trivial. In this section we consider single agents that can perform multiple actions in parallel, and multi-agent systems in which all the agents share the same reward signal. In both cases the choices are made independently by the different branches (possibly in different layers of a hierarchy) of the agent's policy, or by the different agents respectively. From now on we shall just refer to the multi-agent case, but it is understood the same considerations apply to parallel single-agent systems.

What we want to avoid is that while an agent is exploiting the current best policy, therefore making its value converge to the average, some other agent attempts an exploratory move, and the effect of the exploration of one is mixed to the exploitation of the other. In order to prevent that, every agent must be aware that the reward it is going to receive is affected by someone's exploration, therefore it must not let the value of its current best policy decrease. In addition to the reward signal, then, the agents must share the knowledge of which episodes are exploratory, and not decide whether to exploit or explore randomly as in Algorithm 1. An easy way to implement this mechanism, without communication, is to deterministically decide whether each episode is an exploratory one depending on the current episode number. Exploring every n episodes, for instance, would give the value of the eventual new policy n episodes to converge to its average, before a new policy is tried. The system is globally rewarded and if a joint policy gives a reward higher than the current average, it is stored separately and independently by all the agents at the same time. During the exploiting episodes, on the other hand, all the agents know that the reward they are receiving is from the current best joint policy, therefore they are allowed to decrease its value making it converge to the average.

We believe that blending the evaluation of some action with the exploration of others is at the root of the difficulties for Sarsa(λ) on Non-Markovian domains. This is the main reason behind the mechanism of rises and falls, in which we

reward the policy that has returned a high sample, and then lower its value to the average. During such an assessment though, no exploration is allowed, so that the value of the current policy can be reliably estimated. This idea must be implemented globally on the joint policy space on a multi-agent system, by having the agents agree on when it is appropriate to explore, so that no one ruins its current estimation.

4.5 Summary

We have analysed the behaviour of the traditional, direct, RL algorithms, identifying two possible sources of failure in non-Markovian domains: inconsistency during exploration, and interaction of different parts of a policy in the action-value estimation. Then, we have introduced a family of algorithms, Stochastic Search Monte Carlo (SoSMC), for stochastic optimisation in policy space. Our algorithm uses the value function to store an upper bound for each action under the best policy visited, and keeps updating it while the behaviour improves. The upper bound is continually compared to the value of the currently selected action, while the value of the latter converges to the average. As long as the upper bound is lower than the average of the current best action the policy is left unchanged. By not allowing exploration and policy evaluation to mix, we introduce a method of credit assignment suitable for NMDPs, including the case in which multiple actions are executed at the same time.

5

LearnPNP

State Charts are extremely powerful in what they can represent, but not particularly compact. Parallelism among actions can result in a number of states exponential in the number of actions, and using conditions to make different decisions in different situations can lead to a huge number of edges. While this might not be a problem if the plan is automatically generated, it can be particularly tedious when, on the contrary, it is manually written. As mentioned in the introduction, helping the designer to easily convey his knowledge to the learning algorithm is one of the aims of this work. The designer must be able to devise the sketch of a plan, and let the learning algorithm figure out what the agent should do where the human cannot decide. A fundamental aspect in this respect is that not only the designer can be vague in the temporal specification (the sequence of actions) but also on the spacial specification, since the possibility that the model will not completely justify the reward is always taken into account.

In this chapter, we apply the ideas that led to the transformation of a generic plan into a controllable stochastic process to Petri Nets. We work on Petri Nets to overcome the limitation of finite state machines, and to have a framework feasibly applicable in practise in single-agent as much as multi-agent systems.

5.1 Learning in Petri Nets

We extend Petri Net Plans to create a formalism for model-free learning with Petri Nets. Petri Net Plans are a compact way to represent reactive, conditional, parallel plans (cf. Section 2.3). Before defining our new construct to include choice points, and showing how to exploit Petri Nets' expressive power for learning, we discuss the relationship between PNs and State Charts. Then, we introduce our extension of PNP, LearnPNP, and show how to effectively model and learn multi-agent behaviours. We develop an argument similar to the one that led to the definition of the controllable stochastic process for State Charts, applying it to the language based on Petri Nets that provides the foundation to LearnPNP.

5.1.1 Petri Net Plans vs. State Charts

State machines are a subset of Petri Nets and, with the limitations imposed by PNP on the structure of the net, a plan is indeed a state machine with at most $2^{|P|}$ states, where $|P|$ is the number of places. Therefore, the main difference between the two lies in compactness, rather than expressiveness. If the number of actions executed in parallel is small, State Charts can still be more compact than PNPs, as in the examples of Figure 2.6 and 3.2. When the number of actions soars though, and the coordination among actions becomes complex, Petri Net Plans allow to have a clearer view of the synchronisation among actions in both single and multi-agent plans.

The two formalisms also make different assumptions on the interface with the environment. In particular, PNP relies only on the KB, and the events it considers are solely of the type of conditions that become true. On the other hand, State Charts, as discussed in Section 3.1.3, allow different form of perceptions to determine a state change. For this reason, in PNP, all the aspects of the environment that might affect the execution must be included in the description of the state space, which might make keeping track of the changes a little harder. To partially overcome this limitation, later versions of the executor allowed to declare so called *internal conditions* to be tested internally by each action. Thus, an action can trigger its own termination, checking aspects of the environment not included in the Knowledge Base. This way of terminating actions has proved particularly useful for actions whose termination depends on proprioceptive sensors, like reading joints, that pertain to each specific action and are of no interest to the rest of the plan. Adding conditions related to such situations in the main KB would increase

the description of the domain, with no improvement in modelling the knowledge about the environment. Although increasing the number of predicates does not necessarily increase the state space, as the latter depends on the actual abstraction employed by the plan, it does make the design and maintenance of the knowledge base more onerous.

In the rest of this work, we are going to make use of such internal conditions, so that when an action terminating transition has no condition associated, it is implicitly internally terminated. Thus, for instance, the action `Kick` generates itself the event of its termination when the joints reach the target position, without having to declare the condition `kickTerminated`, that would make little sense other than during `Kick`'s execution.

Finally, extending PNP by allowing to break some of the restrictions it makes over Petri Nets, can significantly increase the expressiveness over State Charts. In the next section we define the first extension, that allows the introduction of non-deterministic choices to be controlled by a learning algorithm. The resulting formalism is called LearnPNP and constitutes the main element of the learning framework we developed.

5.2 Introducing non-deterministic choice points

In addition to the basic actions and operators described, we also make use of *non-deterministic choices*, that are structures similar to sensing actions, but not labelled with conditions. Therefore, when the transitions of a non-deterministic choice are enabled, the path chosen by the executor is not determined by the operational semantics, thus leaving the possibility of making an informed choice based on experience. An example of non-deterministic choice is reported in Figure 5.1. In the following we will refer to this structure also as *choice* operator. The `choice` operator connects an input PNP Γ^i and n output PNPs $\Gamma^o_1, \ldots, \Gamma^o_n$, sequencing all of the output networks to the input one, and adding for each of them a new transition. Both the `choice` and `fork` operators transfer the control to a set of networks, but with a fundamental difference: the `fork` operator adds a single transition, which multiplies the number of tokens by the number of output networks, transferring the control to *all* of them and indeed creating new threads of execution; the `choice` operator, on the other hand, has one transition for each output network, and all those transitions are *conflicting* since they would consume a token each, but there can be at most one token in their input places. The latter

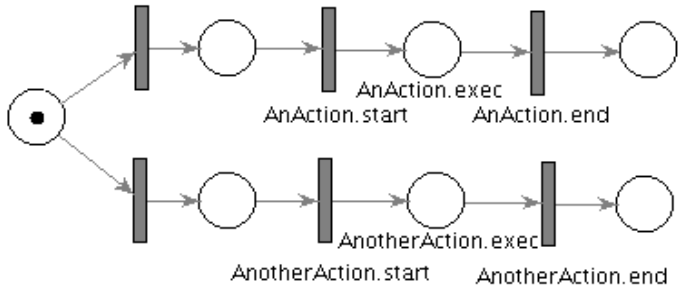


Figure 5.1: An example of a simple choice point between two ordinary actions

case creates the non-determinism.

5.3 The operational semantics

In this section we describe the rules that govern the evolution of such networks, and define under which circumstances a transition is allowed to fire. PNPs differ from ordinary PNs in their operational semantics. The presence of conditions to enable transitions might cause the network to freeze for several time steps, until the knowledge base entails a condition which labels a transition that can fire.

We report here the definitions of a *possible* and an *executable* transition making the role of time more explicit with respect to their original formulation. We consider time in discrete timesteps and actions can take multiple timesteps to complete. As for State Charts, we use the following notation:

- t_k: the time of occurrence of the k^{th} marking transition. By convention we denote $t_0 = 0$
- $M_k = M(t_k)$ where $M(t) = M_k$ for $t_k \leq t < t_{k+1}$ is the marking at time t
- $KB_k = KB(t_k)$ where $KB(t) = KB_k$ for $t_k \leq t < t_{k+1}$ is the state of the Knowledge Base at time t

Definition 5 (Possible transition in a PNP)**.** *Given two markings M_i, M_{i+1}, a transition from M_i to M_{i+1} is possible iff $\exists t \in T$ such that (i) $\forall p' \in P$ s.t $W(p', t) = 1$ then $M_i(p') = 1$; (ii) $M_{i+1}(p') = M_i(p') - 1$ $\forall p' \in P$ s.t. $W(p', t) = 1$; (iii) $M_{i+1}(p'') = 1$ $\forall p'' \in P$ s.t. $W(t, p'') = 1$.*

Definition 6 (Executable transition in a PNP)**.** *Given two markings M_i, M_{i+1} and a Knowledge Base K_i at time t_i, a transition from M_i to M_{i+1} is executable iff $\exists t \in T$, such that a transition from M_i to M_{i+1} is possible and the event condition ϕ labelling the transition t (denoted with $t.\phi$) holds in K_i (i.e. $K_i \models \phi$).*

The executor works at discrete time steps. At every step, it computes the possible transitions, verifies their respective conditions, and fires all the executable ones. Doing so, it calls any function that is associated with the transition, which means it starts, ends, or interrupts actions and plans.

If two or more transitions are executable at the same time but they are conflicting, that is, firing one would prevent the others from firing, it computes all the possible next markings, and delegates the decision to the *learner*. The learner evaluates the possible next markings (for instance through a value function) and makes the choice returning the set of non-conflicting transitions to fire. Then, the reward since the last change in the marking is computed (and opportunely discounted), and fed to the learner so that it can update its estimate of the markings.

At this point, for each execution place active (with a token in it) the executor invokes the corresponding action. If that is not an atomic action but actually a plan, the sub-plan performs the same procedure in turn. A plan might activate more than a sub-plan at the same time, having different sub-plans executing in parallel, and forming a tree of activation (rather then the usual single threaded stack). This way, interrupts in higher-level plans preempt those in lower-level ones. If a plan is terminated the entire sub-tree is terminated downward towards the leaves.

5.4 The learning problem

The central idea of learning in PNP is considering the stochastic process over markings that derives from the semantics of a Petri Net Plan, and learn how to control it in those choice points in which the behaviour is partially specified. In the following we define and analyse such a stochastic process, in an analogous way to what we have done for State Charts, to clarify what we are actually controlling.

5.4.1 Definition of the controllable process

Given a $PNP = \langle P, T, W, M_0, G \rangle$ we define a decision process

$$DP = \langle S, A, Tr, \rho \rangle$$

where:

- $S = \{M_i\}$ is the set of reachable markings executing a sequence of possible transitions from T. This is the state space of our controllable process.
- A is the set of actions. We define an action for each transition introduced by a choice operator, plus one *unnamed* action to account for all the other transitions that there is no need for the controller to distinguish.
- $Tr(s, a, s', \tau) = Pr(t_{k+1} - t_k = \tau, s_{k+1} = s' | s_k = s, a_k = a)$ is the probability for the action a to take τ time steps to complete, and to reach state s' from state s. This probability is defined to be 0 unless the transition corresponding to a is *possible* in the marking corresponding to s and when it fires it transforms such a marking to the marking corresponding to s'. In this latter case, the probability is in general unknown (which accounts for the necessity of learning, otherwise the problem would be solvable by dynamic programming).
- $\rho(s, a, s', k)$ is the reward function, analogous to its counterpart for state charts. Instantaneous rewards are defined over perceptions, that is they are a function of the state of the knowledge base.

Notice that the states for which we learn a policy are a small subset of the reachable markings, being only the markings that make *possible* the few transitions used in choice operators. Both the transition and the reward function depend on the chosen action which is derived from a transition of a Petri Net Plan. If such a transition is not labelled, there is nothing that prevents it from firing when it becomes *possible* and its probability to complete in one time step is 1. On the other hand, if the transition is labelled by a condition the probability of that condition to be true at a specific time step depends on the state of the Knowledge Base and of the environment. The reward might thus depend on the hidden variables of the environment that cause (through the conditions) the transitions to fire. The dependency of the reward from a hidden state can make the process non-Markovian.

5.5 Other possible extensions to PNP

The possibility to partially specify the plan through `choice` operators, and delay those decisions at run time, makes different needs arise with respect to pre-

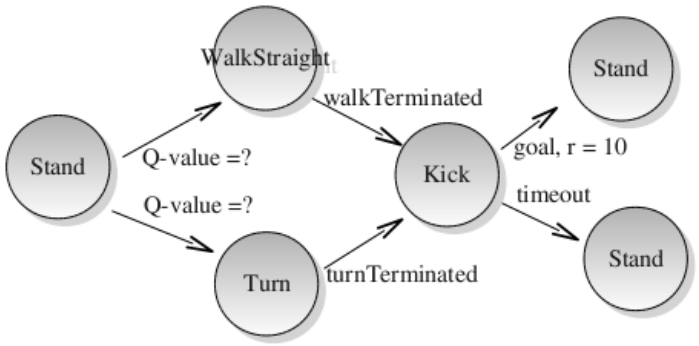


Figure 5.2: Example of a non Markovian reward

encoding everything before the execution. Learning, for instance, may be helped with memory, remembering the previous choices, and it is probably appropriate, in general, to make different decisions in different situations. Petri Nets in this respect prove to be an invaluable tool, and exploiting their representational power the designer can create very compact, clear, and effective structures. In the following we show two possible extensions to PNP, that together with `choice` operators make LearnPNP a powerful tool. Petri Nets allow many different encodings for the same behaviours, so we show a few possible ones with the guideline of keeping the exposition clear. Trickier structures can probably be more compact or more effective, and by no means we intend to limit the designer's degrees of freedom by imposing a particular practise.

5.5.0.1 Memory

Memory can be implemented structurally, in terms of places and transition, in quite the same way as for state machines. For instance a tree structure would store a different Q-value for each branch, remembering each choice. Consider for instance the example of Section 3.2.2 reported for ease of reference in Figure 5.2 Since the choice in `Kick` gives different rewards depending on the action taken at a previous step, this is a case of hPOMDP (cf. Section 1.5). In such domains, keeping track of that choice can help disambiguate the states that provide different rewards. A possible way to do so is turning the plan into a tree, unwinding the structure for all possible choices. This is represented in Figure 5.3 In a tree

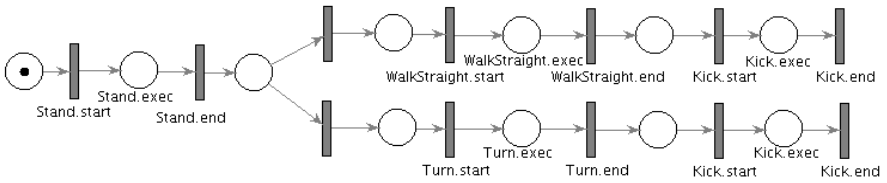


Figure 5.3: A tree structure to disambiguate choices

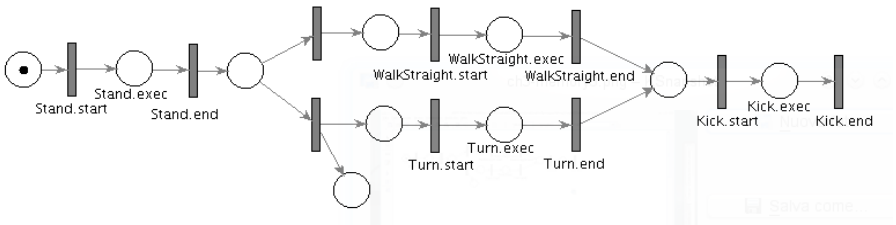


Figure 5.4: A memory place, to remember previous choices

structure the action Kick is duplicated and the two branches receive different rewards, so that when seen from the choice point they do not look the same anymore. Duplicating the action can hardly scale though, and Petri Nets can be used much more carefully.

Recall that we associate a value of the cumulative expected reward to each *marking*, so we can use tokens, instead of duplicating entire portions of the network, to store previous choices. In Figure 5.4 we show the same behaviour as the tree, without duplicating the branches. A `fork` operator is used to create a new token that stores the choice. The marking of the new place is 0 if `WalkStraight` has been chosen and 1 otherwise. When the execution reaches the action `Kick`, it does so with two possible markings, represented in Figure 5.5 and 5.6, therefore along the whole path after the choice point the markings are different depending on the choice, and get rewarded differently. These plans break the assumption made by PNP that each token represents a thread in execution, since no action is associated to the memory places. Moreover when defining goal markings memory must be taken into account, since the route that has led to the goal becomes part of the goal definition. To prevent this from happening a `join` operator, mirroring the `fork` operator that created the memory token, can remove the token before the plan reaches any goal marking. It is also possible to define goal markings considering only a subset of the set of places. When such a subset is in a goal marking the whole network is considered to have reached the goal, regardless of

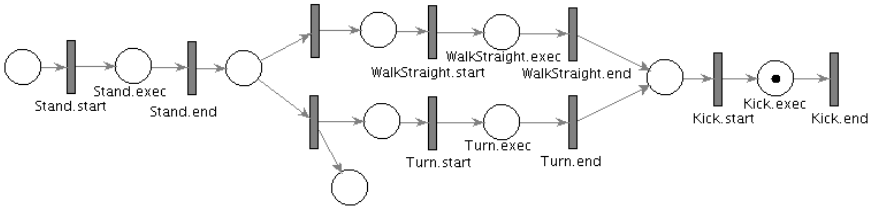


Figure 5.5: Marking after WalkStraight has been executed

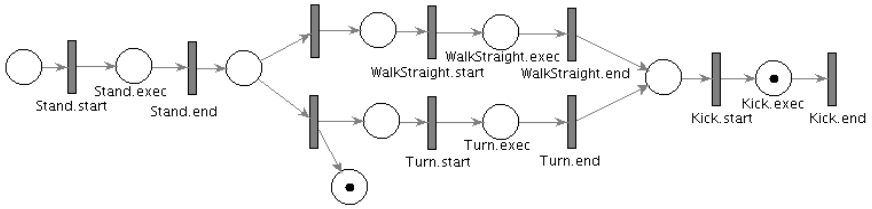


Figure 5.6: Marking after Turn has been executed

the marking of the other places not in that subset.

If we also discard the assumption of 1-boundedness, memory places allow to count loops, in order to make a different choice (and store a separate value) at each pass. This is particularly useful when at the each step the same actions are available. Consider, for instance, a maze with binary junctions. A plan to reach the end should have a choice for each junction. With a state machine, or a traditional PNP, we must have at least a state (or a place) for each junction to associate to it the respective choice. With LearnPNP we can write the plan in Figure 5.7, in which a new marking is created at each iteration, adding a token to the memory place. This still generates a *state* for each choice (a marking in the Petri Net) but the representation is much more compact. Considering expressiveness, this simple network can hold a plan of any finite length, without having to pre-encode a number of choices in the plan. No finite state machine can achieve the same representation without adding states while going. Therefore, in this case the number of choices can be unknown in advance, and the network will generate as many marking as necessary, without having to guess, or bound this number.

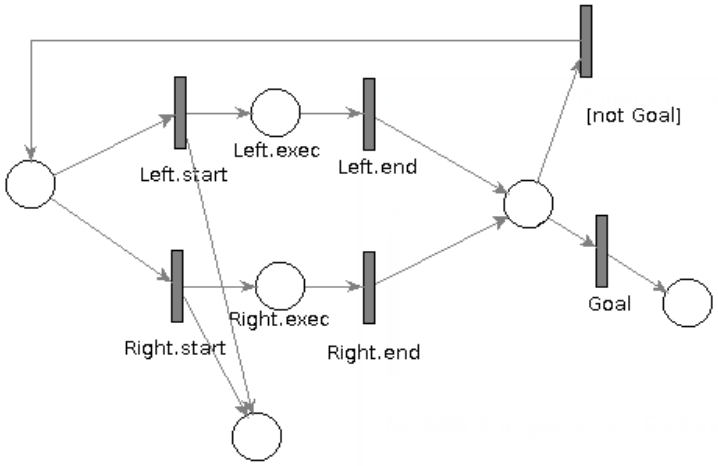


Figure 5.7: A compact way to count loops

5.5.0.2 Environment modelling

In the previous section we have made use of the expressive power of Petri Net to memorise previous choices, and create a more compact structure for time-extended procedures. In this section we show a similar application to the state space. Making different decisions in different parts of the state space is fundamental, even if such an environment can only be partially observable. In PNP the only way to do so is using sensing actions to sort out the part of the space the agent believes to be in, and then choose the corresponding action. Having one token per thread though is quite inefficient, and the number of places required grows high fairly quickly. On the other hand, conditions may be quite effective for factored representations, and again the network itself can generate a marking for each state, with no need to use a place to represent it. Consider for instance a small domain with two binary variables, determining four states. Let the number of actions available in each state be again four. The PNP to make a different decision in every state would look like the one in Figure 5.8. Initially an instantaneous sensing action is performed with the four possible states, and then the actions available in each state are replicated. Unfortunately this structure is impractical even with a small number of states. If we again relax the assumption of one token per thread, and make the network not connected, we can devote a part of the network to track the changes in the environment. The other part of the network keeps being the procedural one, with actions and choice points, but

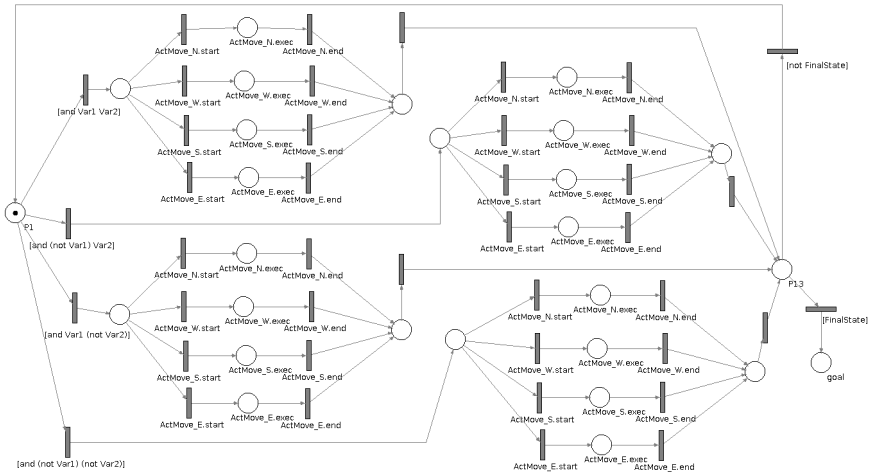


Figure 5.8: A PNP for a domain with four states and four actions

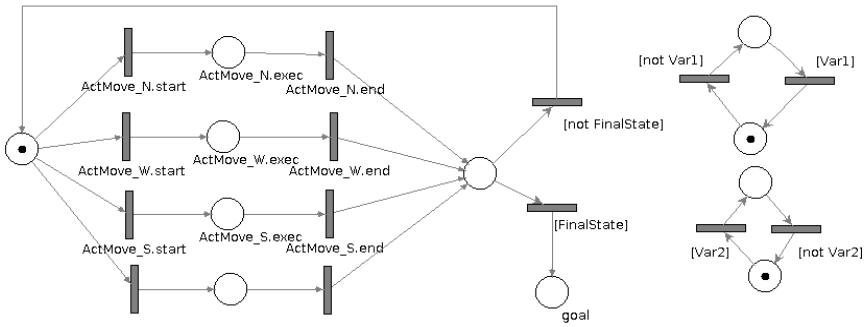


Figure 5.9: A LearnPNP for more compactly represents the same behaviour

gets enormously simplified. The same behaviour of Figure 5.8 can be represented as in Figure 5.9. In this case the right-most part of the network keeps track of change in the environment. The transitions labelled with guard conditions can be considered as instantaneous sensing actions, and if the respective transition is fired the agent believes that condition to be true. At each time step the marking of that portion of the network reflects the knowledge of the agent about the state. Therefore, when the executor is at the choice points and compute the possible next marking they will include that state description and there will be a different marking for each state. There is no need to replicate the actions if they are available in all (if necessary but a few) states.

5.6 Summary

We have introduced LearnPNP, an extension of Petri Net Plans to make use of Petri Nets for learning parallel and multi-agent behaviours. PNP can be effective in designing the synchronisation of actions, but it is not quite suitable for learning as it is, as every choice point should replicate the structure of the choice several times. With the modifications to the formalism proposed in Section 5.5, we can exploit the expressiveness of Petri Nets providing a powerful tool that allows a novel way of modelling and *learning* behaviours under partial knowledge with PNs.

Part III

Experimental Results

6

Grid Worlds

We begin our experimental evaluation of the ideas and algorithms of this dissertation with a simple domain, that can help us compare the results with the literature on non-Markovian environments. We are then going to move to a more complex domain that provides a benchmark for RL methods. In this first part we test our algorithm SoSMC and compare it with Sarsa, that has a similar way of using samples (averaging through a parameter α), and leave the hierarchical modelling and multi-agent evaluation to the next chapter.

6.1 Parr and Russell's Grid World

This small grid world has been used as a test domain by several authors (Parr and Russell, 1995; Loch and Singh, 1998; Perkins, 2002) and provides a simple and structured environment with a reasonable branching factor. It has 11 states (Figure 6.1) in a 4 by 3 grid with one obstacle. The agent starts at the bottom left corner. There is a target state and a penalty state whose rewards are +1 and -1 respectively. Both are absorbing states, that is when the agent enters them the episode terminates. Moreover, after every action the agent receives a reward of -0.04. The actions available in every state are `move north`, `move south`, `move east`, and `move west` which succeed with probability 0.8. With probability 0.1 the

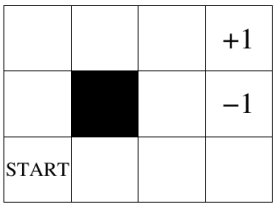


Figure 6.1: Parr and Russell's Grid World

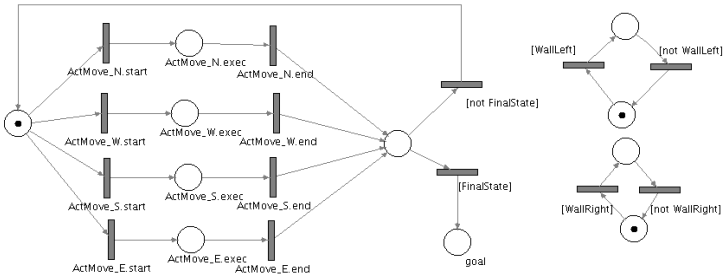


Figure 6.2: The plan for Parr and Russell's Grid World

agent moves in one of the directions orthogonal to the desired one. In all of the previous cases if the movement is prevented by an obstacle the agent stays put. In any state the agent can only observe the squares east and west of it, having a total of four possible observations. Those observations form the state space of an NMDP whose controller we are going to learn.

6.1.1 The Plan

We use LearnPNP to represent a controller for the agent. As described in the previous section there are four actions available in each of the four possible observations. We use the compact representation allowed by Petri Nets, dedicating part of the network to represent the factored aspects of the environment taken into account. The remaining part turns out to be hugely simplified, having just a choice point for the four actions, as shown in Figure 6.2. Recall that as a consequence of Petri Nets' semantics a separate marking will be created at every choice, storing a value for each alternative. We define two conditions, *WallLeft* and *WallRight*, that hold when there is an obstacle to the immediate left and right of the agent respectively. Any marking that has a token in the state labelled with `goal` is considered a goal state, and when entered the episode terminates.

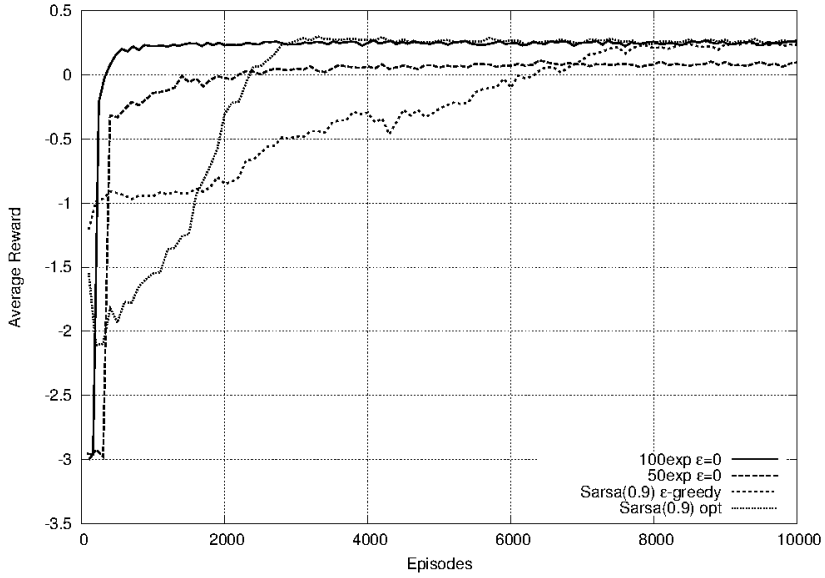


Figure 6.3: Average reward across 200 runs, on the short term for different controllers. SoSMC is evaluated without any exploration in the second phase.

6.1.2 Experimental results

We conducted an evaluation of our algorithm on this grid world, in order to show how the different parameters impact the behaviour of the agent. Every 20 episodes for the short term experiments, and every 100 episodes for the long term ones, we pause the learning and evaluate the current controller for 20 episodes. The results are averaged over 200 runs. By "evaluating the controller" we mean that, during the evaluation, the behaviour of the agent is the same as if it were learning, but the Q-function is left unchanged. Thus, if at a specific point the agent would choose a policy at random, it would do so even during the evaluation. Notice that choosing a policy at random, in this context, is different from following the *random policy*. In the former case the same decision is always made in the same state, while in the latter case each time a state is hit a random choice is made.

We compare our results with two control strategies: Sarsa(λ) with ϵ-greedy exploration, and Sarsa(λ) with optimistic initialisation. The latter strategy consists in initialising the Q-function at an optimistic value for each state-action pair, and exploiting the current estimate at any time. We borrowed a few parameters from the literature (Loch and Singh, 1998; Perkins, 2002) and spent some time optimising others. When not differently specified the Q-function has been initialised at -4. The best behaviour we could achieve for ϵ-greedy was with ϵ starting at 0.2

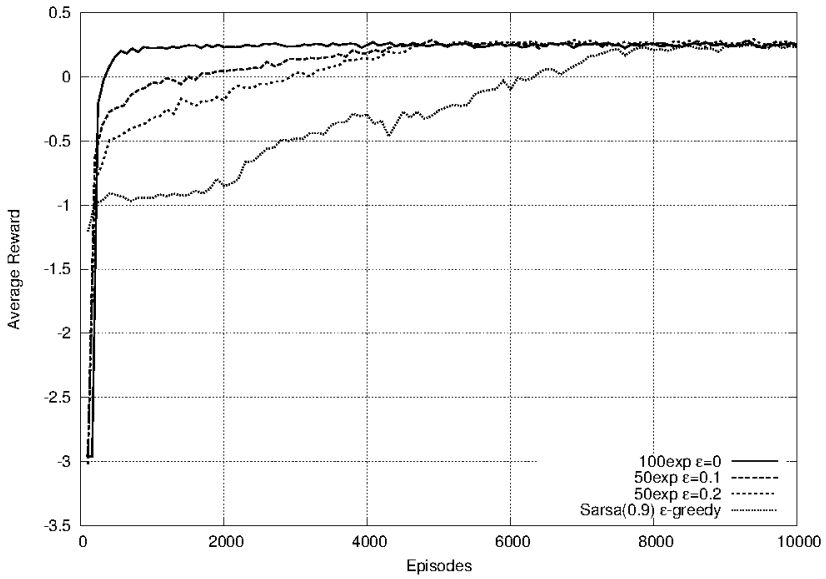


Figure 6.4: Average reward across 200 runs on the long term allowing exploration in the second phase of SoSMC

and linearly decaying to 0 in 80000 actions. For the optimistic initialisation, the Q-function has been initialised at 1. In both cases $\alpha = 0.01$ and $\lambda = 0.9$

Figure 6.3 shows the rewards obtained by different controllers. SoSMC has been evaluated here with ϵ-greedy as its strategy Σ during the first phase, and without any exploration in its second phase. Sarsa(0.9) with optimistic initialisation reliably converges to the optimal policy a lot faster than ϵ-greedy. It is this behaviour that we want to improve, pruning some of the exploration by getting a more realistic initialisation. With an initial phase of 100 episodes and $\alpha = 0.01$, SoSMC always converges to the optimal policy shortly after the initial exploration. We also evaluated the behaviour of the agent with 50 episodes, in order to understand the consequences of little initial sampling. In case the agent could not afford a longer initial phase, we would like that it still quickly converged to a "good" policy, if not the optimal one. Indeed, after 50 episodes the average reward stabilises at around 0.1, while the optimum is around 0.25. Considering that most of the policies give a reward of -4 and that the average reward obtained is non-decreasing, this can be probably considered a good result.

In the second set of experiments we tried to establish whether, by allowing some exploration also on the second phase, it is eventually possible to reach the optimal policy even from a short initial phase. Clearly exploration is a double-

edged sword: on the one hand it allows to discover the optimal policy, on the other hand it worsens the average behaviour of the agent that leaves its current "good" policy. Figure 6.4 shows the results for two different settings, compared with Sarsa(λ) and ϵ-SoSMC after 100 initial episodes as already described. In one setting we let ϵ start at 0.2 and reach 0.01 in 5000 episodes, remaining constant afterwords. In the other setting ϵ started at 0.1. The two results fall in between Sarsa and the optimum obtained with 100 initial episodes. Moreover, the increase in the performance is linear and follows perfectly the decay of ϵ. This probably means that the optimal policy is identified early and, from that point on, the exploration is the only responsible for the sub-optimal behaviour. We have not performed an extensive evaluation over the possible values for the initial ϵ and its decaying rate, therefore we cannot state exactly how close the behaviour can be pushed towards the optimal line above by varying these two values. It seems reasonable though, that the linear dependence allows for a faster convergence up to a point when the exploration becomes too short, and we fall into the initial case of Figure 6.3 with no exploration at all.

6.2 About learning equilibria

We performed several runs to verify the behaviour of Q-learning and Sarsa with a continuous (SoftMax) and a discontinuous (ϵ-greedy) exploration strategy on this non-Markovian domain (cf. Section 1.5.3). We show the value of the four actions in the initial observation during one run for each combination of strategy and algorithm. Both Q-learning (Figure 6.5) and Sarsa (Figure 6.6) showed instability under ϵ-greedy, realising a peculiar cyclic behaviour. Nonetheless, the action selected (the one with the highest value) is the correct one (moving north) most of the time. Under SoftMax both algorithms (Figure 6.7 and Figure 6.8) have been much more stable and their behaviour almost identical. It has proved difficult, however, to find a value for the parameter τ with which they picked the right action. Both graphs show a run with $\tau = 0.01$, while with an higher value the equilibrium reached does not convey any good behaviour. A run with $\tau = 1$ and Sarsa is shown in Figure 6.9

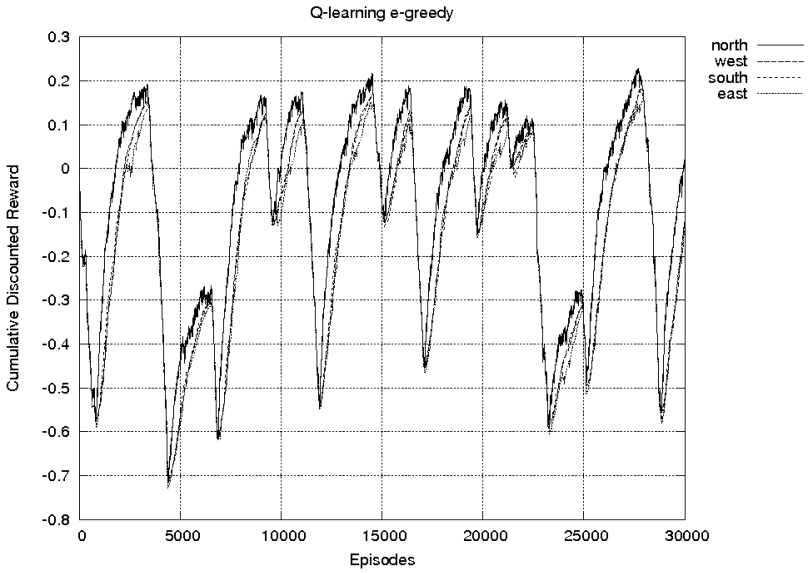


Figure 6.5: Q-learning with ϵ-greedy exploration

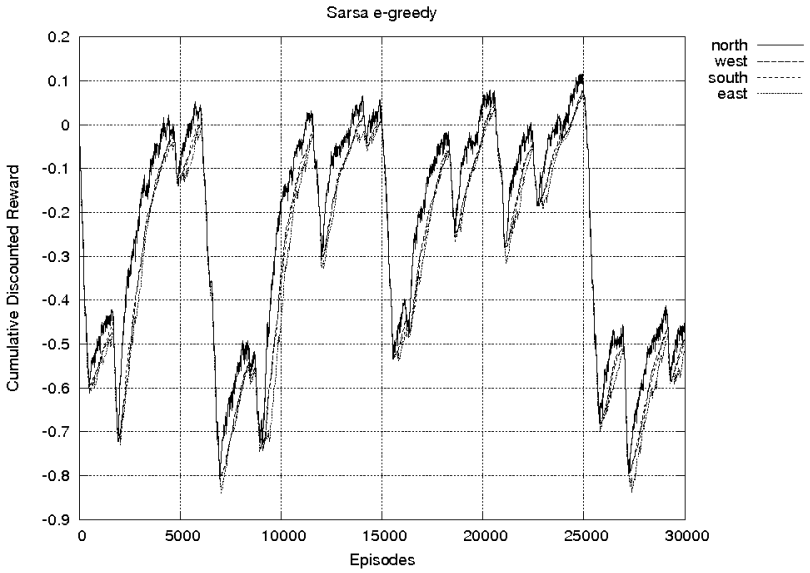


Figure 6.6: Sarsa with ϵ-greedy exploration

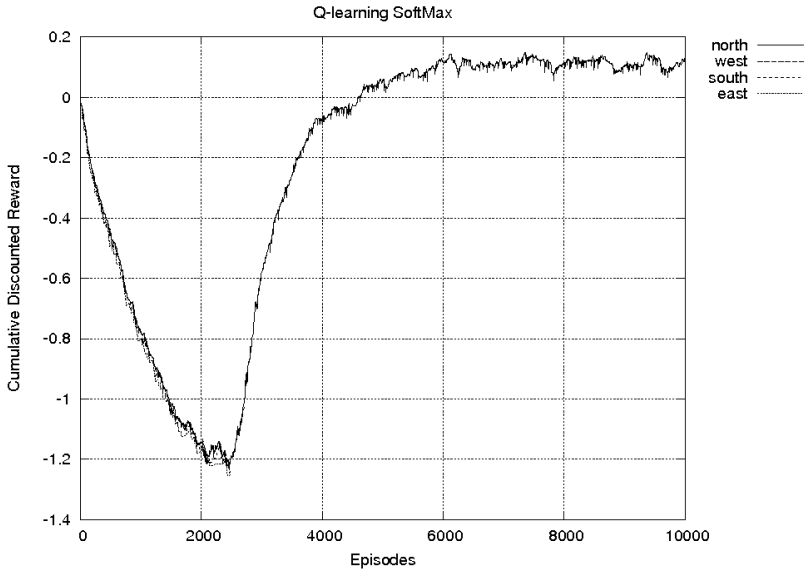


Figure 6.7: Q-learning with SoftMax exploration, $\tau = 0.01$

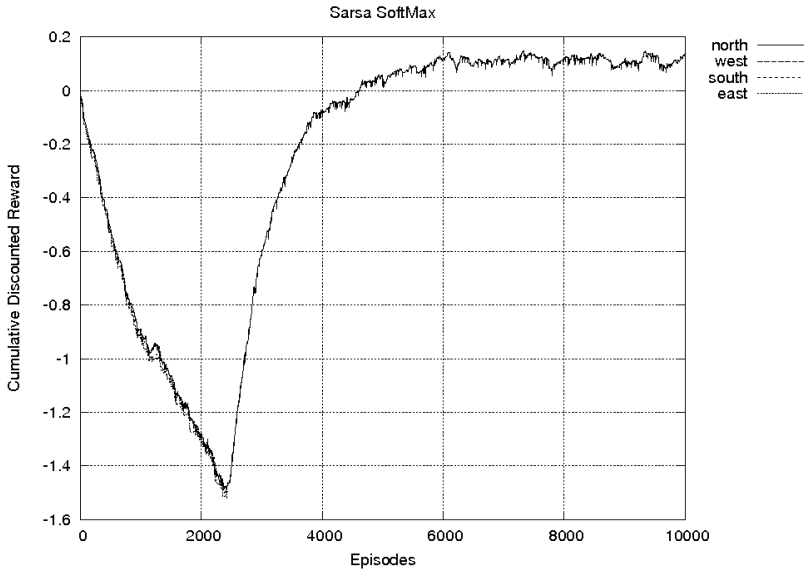


Figure 6.8: Sarsa with SoftMax exploration, $\tau = 0.01$

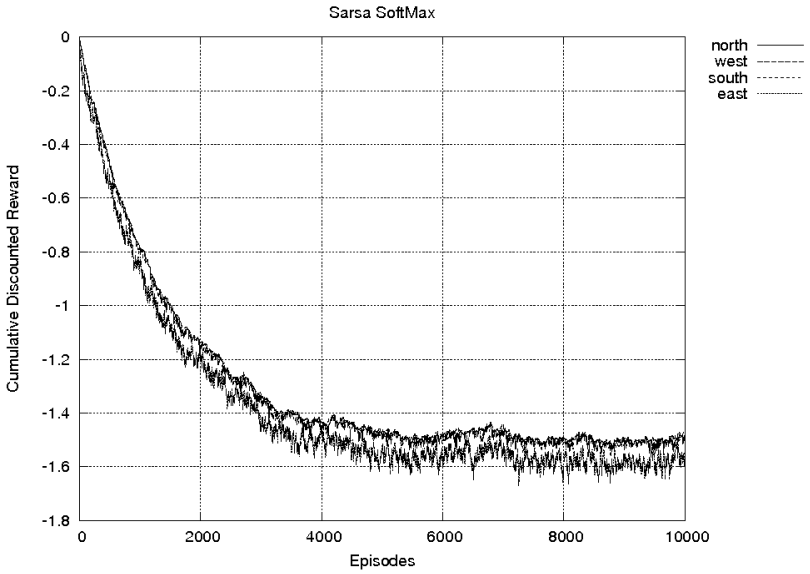


Figure 6.9: Sarsa with SoftMax exploration, $\tau = 1$

6.3 SoSMC and Sarsa more closely

Finally, we show the value function for the same observation-action pair during a run of Soft-SoSMC and Sarsa(λ). The simplicity of this domain allows to concentrate on one choice point, and clearly spot the differences between the two algorithms. We can then discuss the features of SoSMC introduced in Chapter 4. In Figure 6.10, we have explicitly shown a point for each episode, in order to display the respective frequency with which actions are sampled. After the initial exploration phase, the action `move east` obtained a higher reward and therefore looked more promising. Indeed `move east` can be more effective than moving north if, by chance, due to the probabilistic effect of the actions, the agent goes north right after the obstacle. Clearly such a lucky combination is not reliable, and when in the second phase its average is evaluated `move north` prevails. During the assessment phase some exploration is performed as well, and it can be noticed how, according to SoftMax's definition, the actions whose values are closer to the optimal one are tried more often than the others. In particular, `move west`, that indeed makes little sense in the initial observation, is never tried after the first phase. Moreover, the upper bound for `move south` remains far lower than the highest two values, and that action is never chosen during the exploitation.

Figure 6.11 shows a run of Sarsa(λ). Notice how the scale is ten times larger

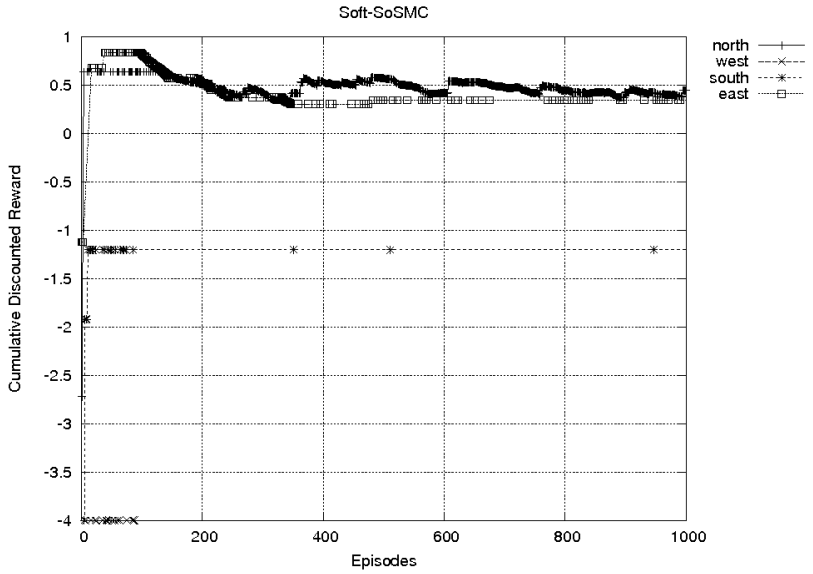


Figure 6.10: SoSMC with SoftMax exploration

than for our algorithm. This run reproduces the results by Loch and Singh (1998). Sarsa is nowadays known to be quite sample-inefficient, and indeed faster algorithms have been developed in conjunction with function approximation (Szepesvári, 2010). With a tabular representation, however, our algorithm is similar to Sarsa in the way of utilising samples, therefore the difference in the performance between the two algorithms is owing to the ideas that led to the definition of SoSMC. We suggest a way of helping the control strategy select the most promising actions, and exploit only those that proved to be so, by continually estimating an upper bound for each action and comparing it with the alternatives. Sample efficiency may be addressed as a future direction, once the features of the algorithm are proved to be effective.

6.4 Sutton's Grid World

Sutton's (1990) grid world is a 9 by 6 grid with several obstacles and a goal state in the top right corner (Figure 6.12). At any given time the agent can observe its 8 neighbouring states, making it a POMDP. Only 30 observations are actually possible in the grid world, and the initial state is chosen at every episode uniformly at random. The actions are the same as the previous domain, but they are deterministic. For this reason, we devised a partially specified behaviour in

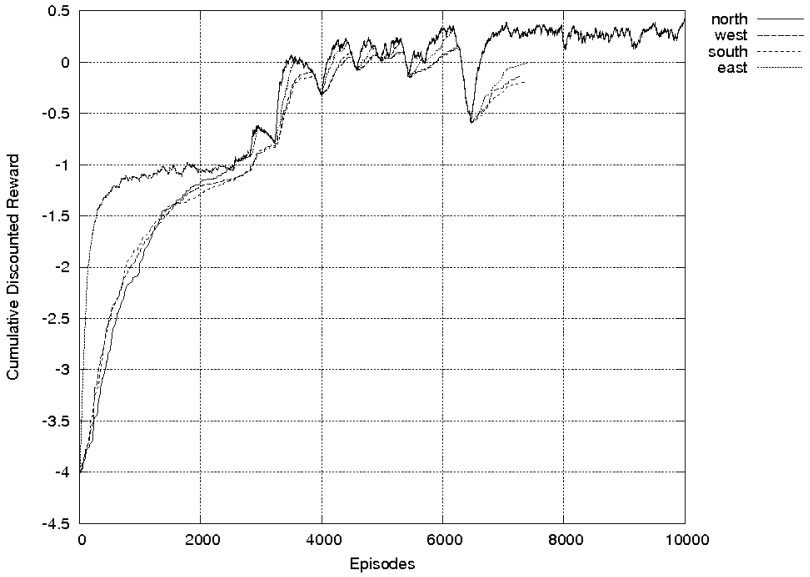


Figure 6.11: Sarsa(λ)

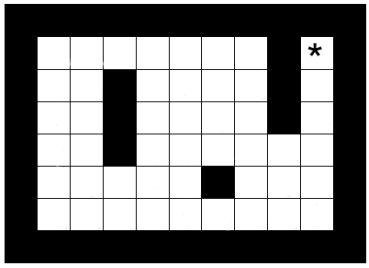


Figure 6.12: Sutton's Grid World

which the agent is allowed to take only those actions that do not lead into an obstacle, as those are certainly useless. The resulting Petri Net Plan is similar to the one for the previous domain, with a larger number of places to take care of the higher number of available perceptions. In order to make the task episodic we set the maximum number of actions in any given episode to 20.

The problem is undiscounted, and after each action the agent receives a reward of -1, except for when it enters the goal state, in which case it is 0. Every 200 episodes we pause the learning and take a sample, from each initial state, of the current best policy, whose average reward per episode is plotted in Figure 6.13.

In this domain ϵ-SoSMC and Soft-SoSMC obtained similar results, therefore we only show Soft-SoSMC. We used SoftMax with no initial phase. The value

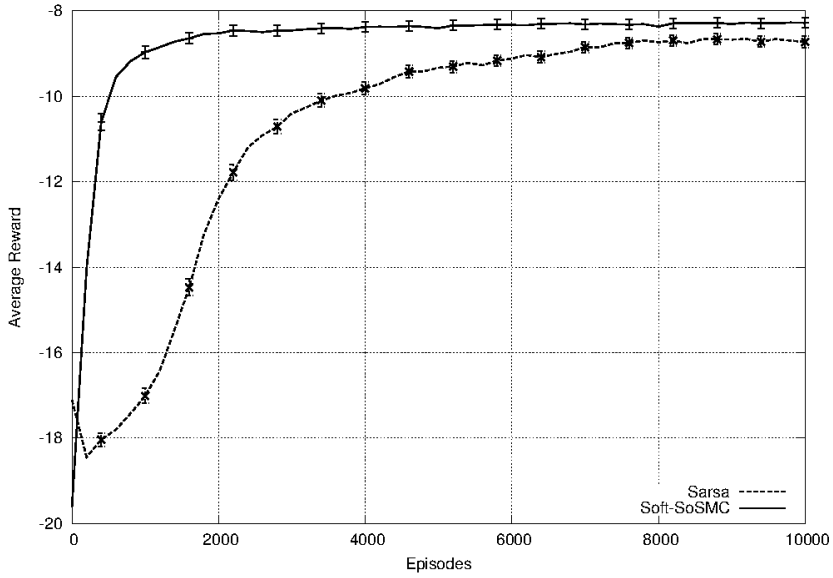


Figure 6.13: Results for Sutton's domain

function has been optimistically initialised and SoSMC launched from its second phase. In this experiment $\tau = 4$ and the algorithm explores every 20 episodes. The results show how Soft-SoSMC finds, on average, a policy better than Sarsa in the number of episodes considered. Sarsa obtains a value slightly, but statistically significantly smaller, as shown by the 95% confidence intervals which have been plotted on the graph every three points in order to not clutter the image.

7

Keepaway

One of the main challenges of RL is scaling to more complex domains. Most of the methods are evaluated on domains synthesised for RL methods, with little practical impact. The number of states is often one of the main concerns, while that is not necessarily the main cause of difficulties in real-world environments. In general, domains specifically tailored for learning help isolate the learning problem from all the other issues that characterise a robotic agent (time consumption, noisy sensors, failing actuators, to name but a few). In complex tasks, however, we expect learning to be seen as part of the system, rather than treated in isolation, and require different solutions for problems that can be very different. The RoboCup (Kitano et al, 1997) provides several domains for both simulated and real robots in different scenarios: home environments, disaster and rescue areas, and football players. Stone et al (2005) proposed a sub-problem of the 2D RoboCup Simulation League as a challenging benchmark for RL methods. We evaluate our framework on this domain, and show a few ways of modelling the problem, with their respective implications.

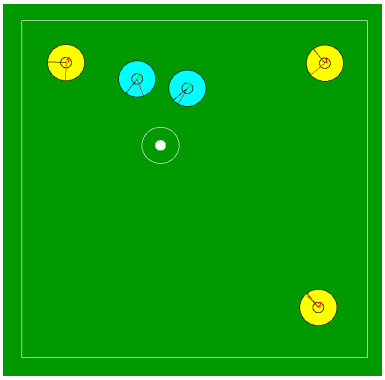


Figure 7.1: The field for Keepaway

7.1 Task Definition

Keepaway is a subtask of RoboCup Soccer in which one team, the *keepers*, must keep possession of the ball in a limited region as long as possible while another team, the *takers*, tries to gain possession. The task is episodic, and one episode ends whenever the takers manage to catch the ball or the ball leaves the region. We conducted our experiments on the 3 vs 2 task, i.e., with three keepers and two takers.

The simulation is performed at cycles of 1/10 of second. Every 1/10 of second the agents can send a command to the simulator. On top of the available commands, two different procedures are defined: `hold` and `pass(k)`. `Hold` keeps possession of the ball until the next decision can be made, while `pass(k)` passes the ball to the k-th team mate, where the team mates are sorted by their distance to the agent. The reward signal r_{t+1} returns the time elapsed since r_t was returned, therefore the sum of rewards equals the global duration of the episode. In the initial formulation, the agents make a decision only when they are the closest one to the ball, and have possession of it. In all other cases, they execute a predefined procedure. We have translated such a procedure into a plan, so that we can write plan schemas for such agents in LearnPNP. The procedural part of the plan for Keepaway is shown if Figure 7.2. Apart from the procedural part there is a network that keeps track of the environment changes, and it depends on what we take into account as our observation space.

The original paper proposes a representation for the state space with 13 continuous variables, accounting for the distances among the agents, the distances among the agents and the centre of the field, and the angles among the agents.

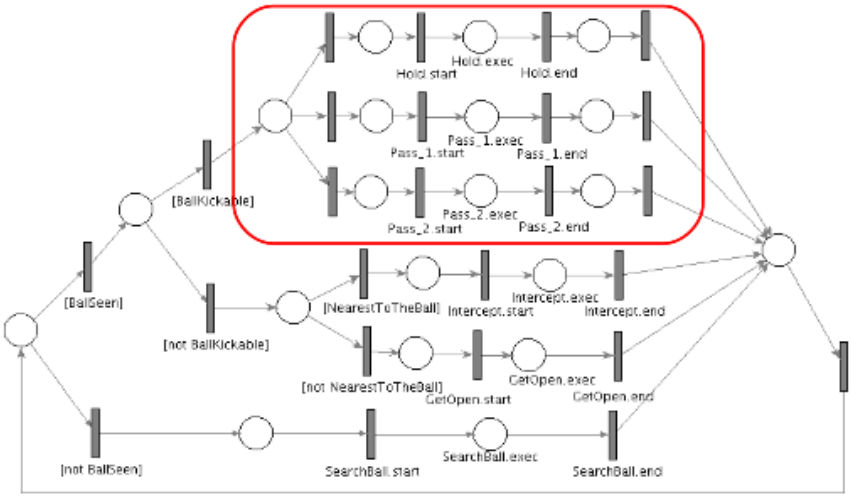


Figure 7.2: The procedural part of the plan for playing Keepaway

Such a representation favours generalisation, being mostly relative to the agent and the centre of the field, and independent of specific locations.

7.2 Single-agent learning

In the experiments shown in this section we have fixed the behaviour of two keepers and learnt the behaviour of the third one. We simplify the representation choosing only three variables: the two distances between the takers and the lines of pass towards the team mates, and the distance between the agent and the closest taker. Moreover, for each variable we consider only one threshold, having 8 observations in total. Certainly these features are not enough to completely justify the reward, leading to a non-Markovian environment. Instead of enriching the representation and generalising from the samples, we pose a different problem, that is, given a (possibly simple) representation, determine the action that performs best across all the possible situations that might occur, even though the representation does not take them into account. The sampling is biased by the actual situations, so that the most common ones impact such an average the most.

The two agents that are not learning wait holding the ball until the takers are closer than a threshold, and then pass the ball to the team mate whose line of pass is farthest from the takers. This simple behaviour, when executed by all the agents, outperforms the currently best results published (Kalyanakrishnan and Stone, 2009), showing that there is indeed hope for simple representations to play

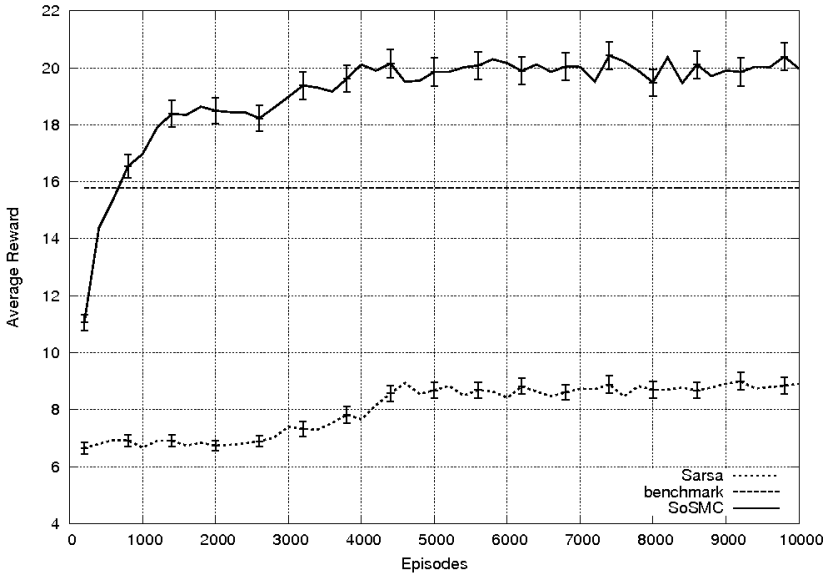


Figure 7.3: Single agent learning the passing behaviour

a role, if an adequate method for learning with them is employed. Clearly, creating such a representation is a separate task which has drawn great attention in recent years. In this work, though, the representation is not particularly sophisticated, and we focus on learning the best behaviour under an incomplete, and inaccurate representation.

We first want to determine whether our algorithm can indeed improve the agent's behaviour where Sarsa(λ) fails to do so. Figure 7.3 shows the reward collected in 10000 episodes averaged over 20 runs by Sarsa(0.9) and Soft-SoSMC. The initial phase of Soft-SoSMC has a length of 100 episodes. While Sarsa(λ) only learns a behaviour that on average holds the ball for 9 seconds, despite the small number of states, our algorithm reaches 16 seconds in less than 1000 episodes and goes up to 20s in the next 4000 episodes. The figure also shows the value of the hand-crafted policy provided with the Keepaway framework, meant to be a benchmark for other methods. Such a policy has, on our system, an average of about 16 seconds, and we have set the threshold for our algorithm at 18s. Therefore, when the performance during the exploiting episodes (exploratory ones are not taken into account) is, on average, higher than 18 seconds across 200 episodes, the exploration is suspended, and resumed only if the reward falls below the threshold.

A future direction, suggested by the ability of learning in these *abstract* states,

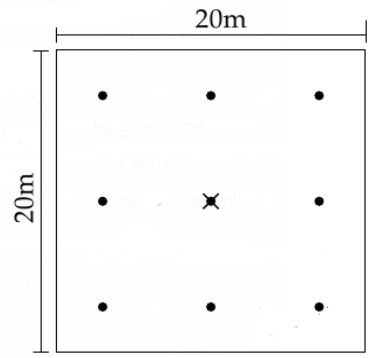


Figure 7.4: Positions available to the agents in the learning task

would be learning at a high level, in which the choice made can be implemented by different strategies (as the decision to pass can be realised by either passing to the closest or to the farthest team mate). This recalls the *angelic semantics* by Marthi et al (2007), in which a high level action has a set of possible realisations among which the agent can choose. With our method, we can estimate an upper bound for the high level actions, possibly connecting hierarchical planning and RL in non-Markovian domains.

7.3 Multi-agent learning

From now on we let all the agents learn at the same time. We consider in this section the problem of positioning, that is, learning a strategy for the agents that are not closest to the ball. We have chosen nine points on the field, as represented in Figure 7.4, and written a plan similar to the previous ones, in which the choice point is on the positioning branch. Each agent executes the same plan, but they separately learn different value functions. In this set of experiments we do not include any state information: the agents just pick a position the first time they have to make a choice and then remain there. The initial state is not always the same, since the simulator places the agents at different corners at random at the beginning. Therefore, they should learn the positions from which the episodes last the longest, across all possible initial states. The three plans executing in parallel can be considered as the second level of a centralised multi-agent plan, since the reward is the same for all of them, but they are executed at the same time making their decision independent.

Figure 7.5 shows the results of the experiment, averaged over 20 runs. Sarsa(λ)

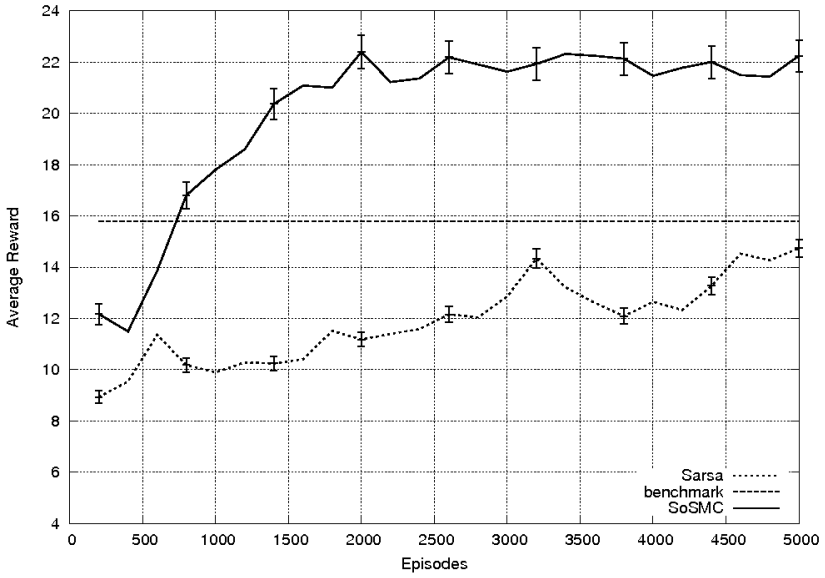
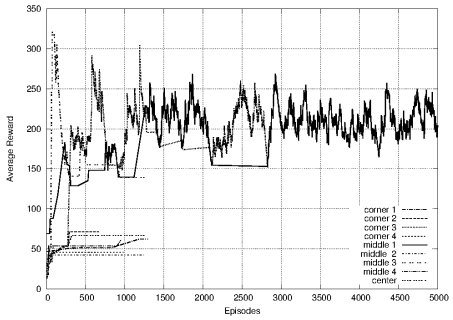


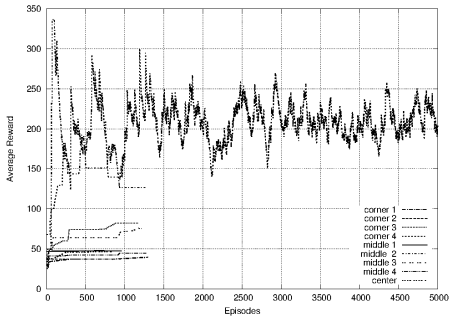
Figure 7.5: Multi-agent Keepaway, learning positioning

with ϵ-greedy, and the degree of exploration decreasing linearly from 0.2 to 0 in 4500 episodes improves the agents' behaviour up to 14.7 ± 0.35 seconds . We have tried several sets of parameters but with no significantly different result. Soft-SoSMC, on the other hand, improves the behaviour quite above the imposed threshold at 18 seconds (which means that the exploration has been suspended at some point).

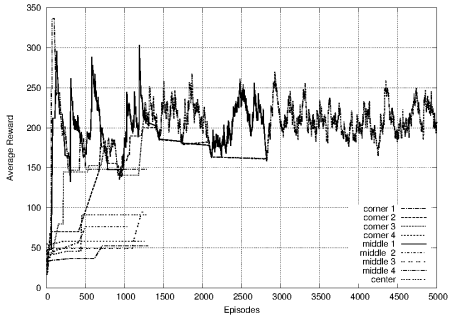
Analysing the achieved behaviour more closely, we noticed that most of the times each agent picks an action and keeps executing it till the end of the epoch, if that action belongs to a policy whose reward is higher than the threshold. Sometimes, however, the team oscillates between two different policies, equivalent in terms of reward. Consider for instance Figure 7.6 which shows the value of all the actions in a particular run. The second agent, when the policy has stabilised over the threshold, executes the action to go to corner 2 (the top-right corner) and never changes. The first and third agent, on the contrary, switch between going to corner 3 (the bottom-right corner) and going to middle point 1 (the leftmost middle point). When one agent does one action, the other one chooses the other action. They never perform the same action that the other agent is doing, and switch a few times between these two equivalent configurations without any communication. Just by sharing the reward function they reach an interesting coordination without the need to communicate.



(a) First agent



(b) Second agent



(c) Third agent

Figure 7.6: The value of each action during a particular run. Note how the first and third agent switch roles between going to corner 3 and middle point 1

8

Conclusions

Reasoning and learning are both characteristic of intelligent agents, but are rarely developed together. We have defined a way in which the two methods can benefit from each other, and make the agent more effective in environments that challenge most of the assumptions of the current frameworks.

We provided a tool that allows designers to let the behaviour autonomously improve, and still be able to revise the choices made by the learner, and modify the plan at an intelligible level. We defined the learning problem on top of a planning process, using the sensing actions of the plan as the only sources of information about the state space. Moreover, we have analysed the resulting controllable process and provided an algorithm to learn in such a context.

In our experimental evaluation, we have shown how a simple behaviour can outperform the one obtained by much more complex representations, although for the latter ones optimality is guaranteed in the limit. Identifying a way to learn such behaviours may then be particularly valuable, as reaching the optimal one is often, in practise, a mirage anyway.

An interesting future direction is moving the same ideas one step further, and apply them to the reasoner rather than to the plan itself. We have shown how a value function can be learnt for fairly high level concepts, such as abstract states. Reasoning at that level can benefit from the experience, and help make a better

use of the samples by recognising similarities, and generalising through inference. Reasoning about what we are learning would be the next step.

Sample complexity is still an issue. In our experiments we have cut the scale of at least one order of magnitude with respect to the previous work, but the number of experiments is still quite high for a real robot. With our algorithm, we have tried to let the performance of the agent be as good as possible even during learning, and more advanced techniques may be pursued to make a better use of each sample.

We discussed how the Markov assumption can actually be a limitation: if on the one hand the results are theoretically comforting, on the other hand, the amount of knowledge required is difficult to achieve. In the attempt to enrich the representation and obtain a globally optimal behaviour, the learning problem can become overly complicated, even for the sophisticated work on generalisation and function approximation. Simpler representations can indeed be more effective, as we have shown on the tested domains, and the conclusion drawn might be less specific and more prone to transfer.

If not as Markovian processes, the domains encountered in real applications should be better understood, and characterised from the agent's point of view. Specific, and theoretically sound, algorithms would then hopefully follow. We like to believe this work can lead to further developments, on both the learning and reasoning side, and make autonomous agents much more effective on the problems where AI still does not seem to scale.

Bibliography

van der Aalst WMP (1998) The application of Petri nets to workflow management. The Journal of Circuits, Systems and Computers 8(1):21–66

Andre D, Russell SJ (2001) Programmable reinforcement learning agents. Advances in Neural Information Processing Systems pp 1019–1025

Andre D, Russell SJ (2002) State abstraction for programmable reinforcement learning agents. In: Proceedings of the National Conference on Artificial Intelligence, pp 119–125

Auer P (2002) Using confidence bounds for exploitation-exploration trade-offs. The Journal of Machine Learning Research 3:397–422

Auer P, Jaksch T, Ortner R (2009) Near-optimal regret bounds for reinforcement learning. In: Koller D, Schuurmans D, Bengio Y, Bottou L (eds) Advances in Neural Information Processing Systems 21, pp 89–96

Barto AG, Mahadevan S (2003) Recent advances in hierarchical reinforcement learning. Discrete Event Dynamic Systems 13(1-2):41–77

Bertsekas DP (1995) Dynamic Programming and Optimal Control. Athena Scientific

Bertsekas DP, Tsitsiklis JN (1996) Neuro-Dynamic Programming. Athena Scientific

Best E, Devillers R, Koutny M (2001) Petri net algebra. Springer-Verlag New York, Inc., New York, NY, USA

Boutilier C, Reiter R, Soutchanski M, Thrun S (2000) Decision-theoretic, high-level agent programming in the situation calculus. In: Proceedings of the National Conference on Artificial Intelligence, pp 355–362

Buşoniu L, Babuška R, De Schutter B, Ernst D (2010) Reinforcement Learning and Dynamic Programming Using Function Approximators. CRC Press, Boca Raton, Florida

Celaya JR, Desrochers AA, , Graves RJ (2007) Modeling and analysis of multi-agent systems using petri nets. In: IEEE International Conference on Systems, Man and Cybernetics, 2007. ISIC., pp 1439–1444

Costelha H, Lima P (2007) Modelling, analysis and execution of robotic tasks using petri nets. In: Proceeding of Interantional Conference on Intelligent Robots and Systems (IROS), pp 1449–1454

Crook PA (2006) Learning in a state of confusion: Employing active perception and reinforcement learning in partially observable worlds. Tech. rep., University of Edinburgh

Crook PA (2007) Learning in a state of confusion: Employing active perception and reinforcement learning in partially observable worlds. PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics.

Dietterich TG (2000) Hierarchical reinforcement learning with the maxq value function decomposition. Journal of Artificial Intelligence Research 13:227–303

Dzeroski S, Raedt LD, Driessens K (2001) Relational reinforcement learning. Machine Learning 43(1/2):7–52

Fikes R, Nilsson N (1971) STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2:189–208

Firby RJ (1989) Adaptive execution in complex dynamic worlds. PhD thesis, Yale

Firby RJ, Prokopowicz PN, Swain MJ (1998) The animate agent architecture. Artificial intelligence and mobile robots: case studies of successful robot systems pp 243–275

Gat E (1992) Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In: Proceedings of the Tenth National Conference on Artificial Intelligence, pp 809–815

Giunchiglia F, Walsh T (1992) A theory of abstraction. Artificial Intelligence 57(2-3):323–389

Harel D (1987) Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3):231–274

Kaelbling LP, Littman ML, Cassandra AR (1998) Planning and acting in partially observable stochastic domains. Artificial Intelligence 101(1-2):99–134

Kalyanakrishnan S, Stone P (2009) Learning Complementary Multiagent Behaviors: A Case Study. In: Proceedings of the 13th RoboCup International Symposium, pp 153–165

King J, Pretty RK, Gosine RG (2003) Coordinated execution of tasks in a multiagent environment. IEEE Transactions on Systems, Man, and Cybernetics, Part A 33(5):615–619

Kitano H, Asada M, Kuniyoshi Y, Noda I, Osawa E, Matsubara H (1997) Robocup: A challenge problem for ai. AI Magazine 18(1):73–85

Konolige K (1997) COLBERT: A language for reactive control in saphira. Lecture Notes in Computer Science 1303:31–50

Konolige K, Myers K, Ruspini E, Saffiotti A (1997) The Saphira architecture: A design for autonomy. Journal of Experimental and Theoretical Artificial Intelligence 9(1):215–235

Kuo CH, Lin IH (2006) Modeling and control of autonomous soccer robots using distributed agent oriented petri nets. In: IEEE International Conference on Systems, Man and Cybernetics, vol 5, pp 4090–4095

Li L, Walsh TJ, Littman ML (2006) Towards a unified theory of state abstraction for MDPs. In: Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics, pp 531–539

Littman ML (1994) Memoryless policies: Theoretical limitations and practical results. In: From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior, pp 238–247

Loch J, Singh S (1998) Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In: Proceedings of the Fifteenth International Conference on Machine Learning, pp 323–331

Loetzsch M, Risler M, Jungel M (2006) Xabsl - a pragmatic approach to behavior engineering. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, 2006, pp 5124–5129

Maier C, Moldt D (2001) Object coloured petri nets - a formal technique for object oriented modelling. Concurrent object-oriented programming and petri nets: advances in petri nets pp 406–427

Marthi B, Russell SJ, Latham D, Guestrin C (2005) Concurrent hierarchical reinforcement learning. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI), pp 779–785

Marthi B, Russell SJ, Wolfe J (2007) Angelic semantics for high-level actions. In: 17th international conference on automated planning and scheduling (ICAPS)

Milutinovic DLP (2002) Petri net models of robotic tasks. In: IEEE International Conference on Robotics and Automation (ICRA'02)

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

Murata T (1989) Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4):541–580

Parr R, Russell S (1998) Reinforcement learning with hierarchies of machines. Advances in neural information processing systems pp 1043–1049

Parr R, Russell SJ (1995) Approximating optimal policies for partially observable stochastic domains. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, pp 1088–1095

Pendrith MD, McGarity M (1998) An analysis of direct reinforcement learning in non-markovian domains. In: Proceedings of the Fifteenth International Conference on Machine Learning (ICML), pp 421–429

Perkins TJ (2002) Reinforcement learning for POMDPs based on action values and stochastic optimization. In: Proceedings of the National Conference on Artificial Intelligence, pp 199–204

Perkins TJ, Pendrith MD (2002) On the existence of fixed points for Q-learning and Sarsa in partially observable domains. In: Proceedings of the Nineteenth International Conference on Machine Learning, pp 490–497

Pettersson O (2005) Execution monitoring in robotics: A survey. Robotics and Autonomous Systems 53(2):73–88

Puterman ML (1994) Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley-Interscience

Rogers D, Plante R, Wong R, Evans J (1991) Aggregation and disaggregation techniques and methodology in optimization. Operations Research 39(4):553–582

Sheng W, Yang Q (2005) Peer-to-peer multi-robot coordination algorithms: petri net based analysis and design. Advanced Intelligent Mechatronics Proceedings, 2005 IEEE/ASME International Conference on pp 1407–1412

Simmons R, Apfelbaum D (1998) A task description language for robot control. In: IROS, Victoria, BC, Canada, vol 3, pp 1931–1937

Singh S, Jaakkola T, Jordan M (1994) Learning without state-estimation in partially observable Markovian decision processes. In: Proceedings of the eleventh international conference on machine learning, pp 284–292

Spall JC (2003) Introduction to Stochastic Search and Optimization, 1st edn. John Wiley & Sons, Inc., New York, NY, USA

Stone P, Sutton RS, Kuhlmann G (2005) Reinforcement learning for RoboCup-soccer keepaway. Adaptive Behavior 13(3):165–188

Sutton R, Barto A (1998) Reinforcement Learning: An Introduction. MIT Press

Sutton R, Precup D, Singh S (1999) Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial intelligence 112(1):181–211

Sutton R, McAllester D, Singh S, Mansour Y (2000) Policy gradient methods for reinforcement learning with function approximation. Advances in neural information processing systems 12:1057–1063

Sutton RS (1990) Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In: Proceedings of the Seventh International Conference on Machine Learning, pp 216–224

Szepesvári C (2010) Algorithms for Reinforcement Learning. Synthesis Lectures on Artificial Intelligence and Machine Learning 4(1):1–103

Viswanadham N, Narahari Y (1992) Performance modeling of automated manufacturing systems. NASA STI/Recon Technical Report A 93:17,572

Watkins C (1989) Learning from delayed rewards. PhD thesis, King's College, Cambridge, England

Wolpert D, Macready W (1997) No free lunch theorems for optimization. IEEE transactions on evolutionary computation 1(1):67–82

Xu D, Volz R, Ioerger T, Yen J (2002) Modeling and verifying multi-agent behaviors using predicate/transition nets. In: SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering, ACM, New York, NY, USA, pp 193–200

Ziparo V, Iocchi L, Lima P, Nardi D, Palamara P (2010) Petri Net Plans. Autonomous Agents and Multi-Agent Systems pp 1–40