Hierarchical Problem Solving using Reinforcement Learning: Methodology and Methods

by

Yassine Faihe

DISSERTATION

Submitted to the Faculty of Science in fulfillment
of the requirements for the degree of

"Docteur ès Sciences"

University of Neuchâtel
Department of Computer Science
Emile Argand 11
CH-2007 Neuchâtel
Switzerland

1999

IMPRIMATUR POUR LA THÈSE

Apprentissage par renforcement pour la résolution hiérarchique de problèmes : méthodologie et méthodes

de M. Yassine Faihe

UNIVERSITÉ DE NEUCHÂTEL

FACULTÉ DES SCIENCES

La Faculté des sciences de l'Université de Neuchâtel sur le rapport des membres du jury,

MM. J.-P. Müller (directeur de thèse), K. Stoeffel, D. Floreano (EPFL), P. Bourgine (Paris) et T. Prescott (Sheffield UK)

autorise l'impression de la présente thèse.

Neuchâtel, le 27 mai 1999

Le doyen:

F. Stoeckli

F. Stoeckli

This thesis is dedicated to my parents,
to whom I owe everything.

Acknowledgements

I am indebted to my advisor, Professor Jean-Pierre Müller for his support and encouragement. While giving me extensive freedom to conduct my research, he has always provided me with useful advice and original ideas. My introduction to the field of *reinforcement learning* as well as the direction taken by my research come from his guidance.

I am grateful to my thesis committee. I greatly acknowledge Paul Bourgine who has helped me to develop the mathematical aspect of my thesis and for the useful discussions we had in Paris. I would also like to thank Tony Prescott for his explanations which have been of great help in my understanding of the *action selection mechanism* as well as for his excellent comments about the dissertation. Thanks must also go to Dario Floreano and Killian Stoffel. Their questions and their remarks allowed me to clarify some important issues.

The interactions I have had with the CASCAD Team members: Antoine, Abdelaziz, Eric, Fabrice, Luba, Luc-Laurent and Matthieu have always been fruitful and of great interest.

Finally I would like to thank Carolina Badii who has proofread the draft of this dissertation and has helped in improving the style of the written English.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context and Motivation

This thesis is about the use of autonomous agents to solve problems. A problem is defined by an environment and a task to achieve. For instance, the environment could be a building with an elevator group and the task could be to control the elevator cars so as to reduce the passengers' waiting time (Crites 1996). An autonomous agent is an entity that has the ability to interact, without human intervention, with dynamic and unpredictable environments through sensing and acting devices. It can sense some aspects of the environment's state and influence its dynamics. During this interaction the agent exhibits a behavior. When tightly coupled with the environment, the agent is said to be embedded (Kaelbling 1993b), that is, being a part of this environment and having quick reactions to stimuli.

The classical approach to building embedded autonomous agents has been to program them. The designer uses his own expertise and *a priori* knowledge to anticipate all possible patterns of interaction, or analyzes and models the problem with differential equations. In the latter case the agent's controller is derived using methods developed in the field of control theory. However the increasing complexity of the problems, coming from difficult tasks or from non-linear, stochastic and unstructured environments, limits the applicability of such methods, even though adaptive methods to tune certain parameters of the controller do exist.

One way of overcoming this difficulty is autonomous programming, that is, making the agent acquire the necessary skills to achieve the given task from the interaction with the environment. Such a process is called learning and refers to the ability to modify one's knowledge according to experience. Apart from freeing the designer from explicitly programming the agent, learning is useful to maintain the agent's capability to perform a task under changing circumstances. Thus learning agents are more flexible, robust and able to cope with uncertainty and changing environments.

First research on learning focused on supervised learning where a tutor trains a system using input-output pairs examples. Because such training examples are not always available, applications of supervised learning methods are restricted to patterns recognition and classification, and functions approximation. Reinforcement learning (RL) is applicable in more general and difficult cases. In the reinforcement learning paradigm, an agent learns how to achieve a given task from its own interaction with the environment. To do so it modifies its decision process on the basis of a feedback which is a scalar evaluation of its current performance. Positive and negative (high and low) values of this scalar correspond to rewards and punishments respectively. Thus the agent solves the problem when it behaves in a way that maximizes rewards and minimizes punishments. RL methods have proven to perform well on simple problems but become impractical to use when the problem's complexity increases.

The main motivation of the work presented in this dissertation is to scale up reinforcement learning to complex problems.

1.2 Claims and Proposals

Two closely linked reasons can explain why reinforcement learning fails to solve complex problems. First the appropriate reinforcement function, that is, the one that makes the agent solve the problem when rewards are maximized, is not easy to find. So far there has been no systematic way to design such a function. The second reason is that the number of situations that the agent may encounter during its interaction with the environment increases with the complexity of the problem, so the search process is slowed down and becomes complicated. This phenomenon is called the *curse of dimensionality.*

We claim that a good understanding of the difference between a behavior and the mechanism that produces it, as well as the underlying consequences, will provide useful insights to overcome the above difficulties. We argue that:

- a behavior is the description, from an external observer's point of view, at different levels of abstraction, of a sequence of actions produced by the agent via its coupling with the environment;
- complex behaviors may be produced by the coordination of several simple sensory-motor mechanisms interacting with the environment (Braitenberg 1984; Pfeifer and Scheier 1998);
- solving a problem using an embedded agent amounts to designing the corresponding behavior;
- the design process of a behavior consists in transposing the observer's point of view into the agent's point of view;

Having these arguments in mind, it is now possible to tackle the obstacles that limit the scalability of reinforcement learning.

Let's start with the curse of dimensionality. When a problem requires to be solved in which the agent performs a long sequence of actions, it becomes very hard to discover such a sequence, especially when the reinforcements are sparse because the exploration is not guided. One may introduce local reinforcements (given by a teacher) to guide the exploration or come up with efficient exploration strategies. One may also argue that the agent does not have the adequate actions otherwise it would have solved the problem in few decision steps (Martin 1998). Thus, we propose to add the missing actions to the agent repertoire by allowing it to learn them. Actually these new actions correspond to skills that solve parts of the problem. So it is necessary to perform a problem decomposition in order to identify the needed skills. If the skills found are still too difficult to learn, the corresponding sub-problems are decomposed once again. The resulting agent's architecture is a hierarchically structured skills set where each skill is learned using previously acquired ones.

The direct consequence of this approach is that we will have to design several simple reinforcement functions (one for each sub-problem) rather than a single global and complex one. However the necessity to have a means of describing behaviors still remains.

In order to systemize the approach we mentioned above, and manage the overall design process a methodology is required. Issues that should be raised by such a methodology concern:

- the analysis of the problem and the specification of the desired behavior;
- the problem decomposition into sub-problems and the learning of the corresponding skills;
- coordination of these skills to solve the global problem.

A methodology that meets these requirements as well as methods to address the above issues are proposed in this thesis, and constitute our main contribution.

1.3 Organization of the Dissertation

In this thesis we investigate the methodological aspect of hierarchical problem solving using agents that learn by reinforcement. The next chapter defines the reinforcement learning problem. It provides a mathematical formulation of the problem and reviews techniques to solve it. Chapter 3 presents the postman robot problem and describes the testbed used in this work. In chapter 4 a new agent design methodology is introduced with details of its components. One particular component of the methodology, the coordination, is addressed in depth in chapter 5. Both chapters 4 and 5 report and analyze the experimental results we have obtained. Finally in chapter 6, we summarize the contribution of our work, discuss some practical issues, and suggest directions for future research.

Chapter 2

Background: Reinforcement Learning

In this chapter we introduce the reinforcement learning problem. We first setup the framework by defining how the agent interacts with the environment and formalize the problem as the optimal control of a Markov decision process. The solutions are presented from the credit assignment point of view. Both temporal and structural credit assignment problems are described and state-of-the-art methods to solve them are reviewed.

2.1 Formulation

2.1.1 Framework

The *agent*, the *environment* it interacts with and the *task* it has to achieve are the components that define the reinforcement learning framework (figure 2.1). The interaction between the agent and the environment is continuous. On one hand the agent's decision process selects *actions* according to the perceived situations of the environment, and on the other hand these situations evolve under the influence of the actions. Each time the agent performs an action, it receives a *reward*. A reward is a scalar value that tells the agent how well it is fulfilling the given task. To be formal let's denote x a *representation* of the environment's *state* as it is perceived by the agent, a the selected action, and r the received reward. The agent's decision process is called *policy* and is a mapping from states to actions. A learning agent modifies its policy according to its experience and to its goal which is to maximize the cumulated rewards over time. Such an amount is called *return*

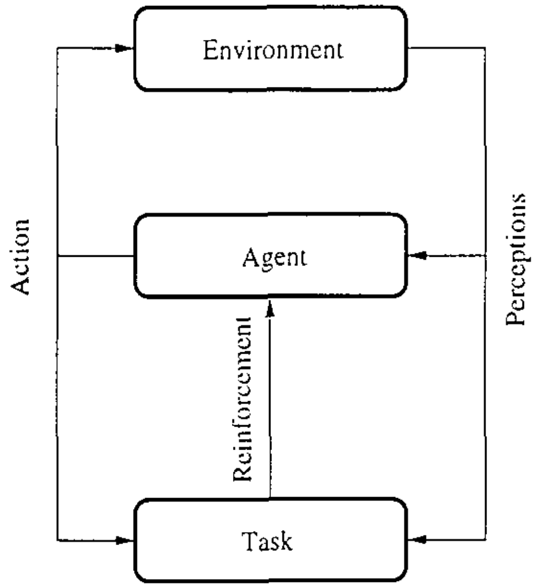

Figure 2.1: Reinforcement learning framework

and will be explained later. Because of its flexibility and its abstraction, the reinforcement learning framework can be used to specify several kinds of problems. Actually, time steps at which an interaction occurs have to be seen as decision-making steps rather than fixed ticks of real time, and states and actions may range from low-level interaction devices to high-level descriptions and decisions.

2.1.2 Markov Decision Processes

A *Markov decision process* (MDP) consists of a set of states X and a set of actions A which allow movement from one state to another. In each state x only a subset of actions $A(x) \subseteq A$ is available. The dynamics of the process is governed by a set of transition matrices. There is one matrix $P(a)$ for each action a, where each element $P_{xy}(a)$ denotes the probability of transition to state y given x and a. If an action a is not available in state x then $P_{xy}(a) = 0$. At the end of each transition a reward $r = R(x, a, y)$ is generated. The immediate evaluation of a transition is generally expressed by the expected reward:

$$\begin{aligned} R(x, a) &= E[R(x, a, y)] \\ &= \sum_{y \in X} P_{xy}(a) R(x, a, y). \end{aligned} \tag{2.1}$$

In this thesis we assume that the process is discrete and that both S and A are finite.

Policy

A policy is a mapping $\pi^t : X \mapsto A$ which associates an action a to each state x. We notice that a policy not only depends on the state of the process but also on the time step where the decision is made. Here we will focus on policies that specify actions as a function of a state only. Such policies are called *stationary* and are denoted π.

Markov Property

In general the outcome of a process, in terms of states and rewards, at a given time step depends on the prior sequence of states or past history $H_t = \{x_t, a_t, x_{t-1}, a_{t-1}, ..., x_0, a_0\}$. When it is possible to predict the next state and the next expected reward only on the basis of the current state, then the process is said to have the Markov property or to be Markovian. Formally the Markov property can be expressed by the following equality:

$$Pr(x_{t+1} = x, r_{t+1} = r | H_t) = Pr(x_{t+1} = x, r_{t+1} = r | x_t, a_t) \tag{2.2}$$

One can notice the importance of the Markov property in the sense that the decision is only a function of the current state. The case where an agent has to deal with non-Markov states, either because it interacts with a non-Markov environment or because of its incomplete perceptions, will be discussed later.

2.1.3 Returns and Optimality Criteria

An MDP controlled by a policy π generates a sequence of rewards $R_\pi = \{r_1, r_2, r_3, ..., r_n, ...\}$. To order different policies we can define an *optimality criterion* on this sequence of rewards. Roughly speaking an optimal policy optimizes the total amount of rewards generated over a long run period:

$$r_1 + r_2 + r_3 + ... + r_n + ... \tag{2.3}$$

Such a measure of long term reward is called *return* (Barto et al. 1990). Because of the stochasticity of the controlled process we will consider the expected value of the return.

Moreover we introduce the following generic notation for the return:

$$E_\pi\left[\sum_{t=0}^{N}\omega(t)r_t\right], \tag{2.4}$$

where E_π is the expectation operator when policy π is used, N is the *horizon* of the return and ω is a weighting factor. Several optimality criteria have been investigated in the literature (Mahadevan 1996), but all can be expressed in the above form. Here we will focus on the case where $N \to \infty$ and $\omega(t) = \gamma^t$, where $0 \leq \gamma < 1$, which represents the *expected discounted total reward.* The discount factor acts as an attenuator. Hence one unit of reward received at time $t + \tau$ is equivalent to γ^τ units at time t. This optimality criterion is attractive because of its mathematical properties which make the computation of the optimal policy more tractable: the return value is finite (because $0 \leq \gamma < 1$ and as long as the reward function is bounded) and the optimal infinite horizon policy is always stationary.

2.2 Temporal Credit Assignment

The *temporal credit assignment* problem (TCA) consists in attributing credit or blame to individual actions on the basis of the result of a whole plan of actions and is a concern for most real decision problems. Indeed some actions may generate low immediate payoff but can contribute to producing higher rewards in the future. Sometimes several actions have to be performed before getting a reward: the reward is said to be *delayed.* In this section we review *dynamic programming* (DP) and *temporal difference* (TD) learning which are techniques that solve the TCA problem. Although DP algorithms can compute optimal policies for MDPs, they are not very useful to solve reinforcement learning problems because an accurate model of the environment is usually not available. However dynamic programming provides important theoretical foundations for understanding the function of temporal difference methods.

2.2.1 Value Functions and Optimal Policies

A widely used approach to deal with delayed rewards is to estimate the worth of a state or a decision in terms of future expected rewards. Given an optimality criterion we can define a *value function* for a policy π, $V^\pi : X \mapsto I\!R$ as a mapping from states to real values. We have:

$$V^\pi(x) = E_\pi\left[\sum_{t=0}^{\infty} \gamma_t r_t | x_0 = x\right], \tag{2.5}$$

which expresses the expected return when the policy π is followed starting from state x. In the same way we can define a *utility function* for policy π, $Q^\pi : X \times A \mapsto I\!R$ mapping state-action pairs to real values. $Q^\pi(x, a)$ expresses the utility to perform action a in state x and follow policy π thereafter:

$$Q^\pi(x, a) = E_\pi\left[\sum_{t=0}^{\infty} \gamma_t r_t | x_0 = x, a_0 = a\right]. \tag{2.6}$$

Given two policies π_1 and π_2, we say that π_1 is better than (or an improvement of) π_2 if the value function for the first policy is at least equal to that of the second policy, and is greater for at least one state. Hence the optimal policy π^* is the one which cannot be improved anymore. Its value function is V^*. Many optimal policies may exist but they all have the same optimal value function V^*. Now we will see how such optimal policies can be induced.

2.2.2 Dynamic Programming

The starting point of dynamic programming comes from equation 2.5 written in a recursive form:

$$V^\pi(x) = R(x, \pi(x)) + \gamma \sum_{y \in X} P_{xy}(\pi(x)) V^\pi(y), \tag{2.7}$$

which, for the optimal policy π^*, becomes:

$$V^{\pi^*}(x) = R(x, \pi^*(x)) + \gamma \sum_{y \in X} P_{xy}(\pi^*(x)) V^*(y). \tag{2.8}$$

As all optimal policies have the same optimal value function V^*, and $V^* \geq V^{\pi_i}$ for all $x \in X$ and for all policies π_i, we obtain:

$$V^*(x) = \max_{a \in A(x)} \left[R(x,a) + \gamma \sum_{y \in X} P_{xy}(a) V^*(y) \right]. \tag{2.9}$$

This equation is known as the *Bellman's optimality equation* (or Bellman's equation for π^*). When V^* is known, the optimal policy can be easily derived:

$$\pi^*(x) = \arg \max_{a \in A(x)} \left[R(x,a) + \gamma \sum_{y \in X} P_{xy}(a) V^*(y) \right]. \tag{2.10}$$

There are several computational techniques to solve the Bellman's equation. Here we will limit ourselves to two of them: *value iteration* and *policy iteration*. But let's first see how the evaluation of a given policy can be computed.

Policy Evaluation

Let's define $V_n^\pi(x)$ as the expected return if policy π is followed for n steps only, starting from state x. For $n = 1$, the expected return is simply the expected immediate reward, when action $a = \pi(x)$ is performed:

$$V_1^\pi(x) = R(x,a). \tag{2.11}$$

Assuming that V_1^π is known and that the next observed state when a is performed in x is y with probability $P_{xy}(a)$, we have for all $x \in X$:

$$V_2^\pi(x) = R(x,a) + \gamma \sum_{y \in X} P_{xy}(a) V_1^\pi(y). \tag{2.12}$$

Similarly we can determine V_3^π from V_2^π, V_4^π from V_3^π, and in the general case V_{n+1}^π from V_n^π:

$$V_{n+1}^\pi(x) = R(x,\pi(x)) + \gamma \sum_{y \in X} P_{xy}(\pi(x)) V_n^\pi(y), \tag{2.13}$$

for all $x \in X$. After a high number of iterations N over all states, $V_N^\pi(x)$ can be considered as a good approximation of $V^\pi(x)$ given an arbitrary initial $V_0^\pi(x)$.

Policy Iteration

The policy iteration method consists of two procedures: the policy evaluation and the policy improvement. Thus starting from any initial policy π_0 we will successively evaluate it, obtaining V^{π_0}, improve it, obtaining π_1, and so on until the optimal policy π^* is reached (figure 2.2). Once a policy π_n is evaluated, the result V_n is used to make the improvement.

$$\pi_0 \xrightarrow{\text{PE}} V^{\pi_0} \xrightarrow{\text{PI}} \pi_1 \xrightarrow{\text{PE}} V^{\pi_1} \;\ldots\ldots\; \pi^* \xrightarrow{\text{PE}} V^*$$

Figure 2.2: The policy iteration method build a sequence of policies that converge to π^*. PE and PI are respectively the policy evaluation and the policy improvement operators.

The following update is applied for all $x \in X$:

$$\pi_{n+1}(x) \leftarrow \arg\max_a \left[R(x,a) + \gamma \sum_{y \in X} P_{xy}(a) V_n^\pi(y) \right]. \tag{2.14}$$

Figure 2.3 shows the policy iteration algorithm.

$\pi \leftarrow$ arbitrary policy
$V \leftarrow$ arbitrary function
repeat
  Policy evaluation
  repeat
    for each $x \in X$ **do**
      $V(x) \leftarrow R(x, \pi(x)) + \gamma \sum_{y \in X} P_{xy}(\pi(x)) V(y)$
    end for
  until $\max_{x \in X} |V_n^\pi(x) - V_{n-1}^\pi(x)| < \epsilon$

  Policy improvement
  for each $x \in X$ **do**
    $\pi(x) \leftarrow \arg\max_a \left[R(x,a) + \gamma \sum_{y \in X} P_{xy}(a) V(y) \right]$
  end for
until π is stable

Figure 2.3: The policy iteration algorithm

Value Iteration

The policy evaluation phase in the policy iteration algorithm needs a lot of computation and has to be performed after each improvement. Instead of making an improvement after each policy evaluation, it is possible to make it after only one backup of each state. This procedure amounts to directly compute the optimal value function using equation 2.9. The backup operation becomes:

$$V_{n+1}(x) = \max_a \left[R(x, a) + \gamma \sum_{y \in X} P_{xy}(a) V_n(y)) \right], \tag{2.15}$$

for all $x \in X$. The complete value iteration algorithm is given in figure 2.4.

$V_0 \leftarrow$ arbitrary function

Compute optimal value function
repeat
  for each $x \in X$ **do**
    $V_{n+1}(x) \leftarrow \max_a \left[R(x, a) + \gamma \sum_{y \in X} P_{xy}(a) V_n(y) \right]$
  end for
until $\max_{x \in X} |V_{n+1}(x) - V_n(x)| < \epsilon$

Compute optimal policy
for each $x \in X$ **do**
  $\pi(x) \leftarrow \arg\max_a \left[R(x, a) + \gamma \sum_{y \in X} P_{xy}(a) V_{n+1}(y) \right]$
end for

Figure 2.4: The value iteration algorithm

Asynchronous Dynamic Programming

The algorithms presented in the previous section are called *synchronous dynamic programming algorithms* because at each iteration the value function is updated for the entire state space. In the case where the state space is very large, the solution of the MDP becomes computationally intractable. *Asynchronous dynamic programming* relaxes this rule and allows backups to be applied for only a subset of the state set, which may be a singleton

(Gauss-Seidel DP) and may vary in each iteration. Let $X_n \subseteq X$ be the set of states whose value functions will be backed up during the iteration stage $n = 0, 1, \ldots$ The backups are done as follows:

$$V_{n+1}(x) = \begin{cases} \max_a \left[R(x,a) + \gamma \sum_{y \in X} P_{xy}(a) V_n(y)\right] & \text{if } x \in X_n, \\ V_n(x) & \text{otherwise.} \end{cases} \tag{2.16}$$

The choice of X_n is crucial for the convergence to V^*. Ideally each state should be backed up infinitely, which means that it should be contained in all the subsets X_n.

Adaptive Real-Time Dynamic Programming

The relaxation introduced by asynchronous DP is very useful when the computation of the optimal policy occurs while interacting with an unknown process. In this case the states are backed up as they are encountered. Adaptive real-time dynamic programming (ARTDP) (Barto et al. 1995) relies on this principle to perform an on-line control of a process. It involves the estimation of the process' model, the policy computation, and the control. Each time a transition is observed the estimate of the transition probabilities matrixes $\hat{P}(a)$ is updated:

$$\hat{P}_{xy}(a) = \frac{n_{xy}(a)}{n_x(a)} \tag{2.17}$$

where $n_{xy}(a)$ is the number of transitions from x to y when a is performed, and $n_x(a) = \sum_{y \in X} n_{xy}(a)$ is the number of times a was performed in x. The estimation of the immediate reward $\hat{R}(x,a)$ is simply updated with the average of the observed immediate reward for this state-action pair. After an infinite number of updates the estimated model of the process converges to the true process. At each time step t the optimal value function is estimated using the current process model estimation and the previous optimal value function estimation $\hat{V}^*_{t-1}$. With an accurate model only one backup would be necessary and $\hat{V}^*_t$ would be equal to V^*. However, in the present case such a model is not available and there are little variations between two consecutive estimations of the model. For these reasons active exploration mechanisms have been investigated (Barto and Singh 1990) to speed up the identification phase.

2.2.3 Temporal Difference Learning

Temporal difference learning (Sutton 1988) methods are concerned with solving a prediction problem and unlike DP methods, do not need a model of the environment's dynamics. Such methods are referred to as *direct* or *model-free* methods as opposed to *indirect* methods like ARTDP or *model-based* methods like DP. In this section we present the general principle behind the prediction of the value function of an MDP and then extend it to the control problem. Finally we will see how the efficiency of TD methods can be improved with eligibility traces and review some popular TD algorithms.

Prediction

For a Markov decision process and a policy π, the prediction problem concerns the value function V^π. Let $\hat{V}^\pi(x)$ be an estimate of $V^\pi(x)$. Given an experience $\langle x, a, r, y\rangle$ and the estimates of each of these states, $\hat{V}^\pi(x)$ and $\hat{V}^\pi(y)$, it appears , relying on equation 2.7 that $r + \gamma\hat{V}^\pi(y)$ is a better estimate of $V^\pi(x)$ than $\hat{V}^\pi(x)$. The temporal difference error (TD-error)

$$\Delta\hat{V}^\pi = r + \gamma\hat{V}^\pi(y) - \hat{V}^\pi(x) \tag{2.18}$$

is simply the difference between these two estimates, and is used to update the previous estimate of V^π. The construction of an estimate of V^π directly from the observation of successive states and rewards is done using the following update rule:

$$\hat{V}^\pi(x) \leftarrow \hat{V}^\pi(x) + \alpha\Delta\hat{V}^\pi, \tag{2.19}$$

where $0 < \alpha \leq 1$ is the learning rate. Equation 2.19 is known as the TD(0) equation. Each time the state x is visited and the above update is applied, the estimate $\hat{V}^\pi(x)$ becomes closer to $V^\pi(x)$.

Control

To use TD methods for the control problem, the prediction has to be made on the utility function $Q^\pi(x, a)$ rather than on the value function $V^\pi(x)$. On the other hand we need to expand the experience mentioned above by adding b which is the chosen action when y is observed. At the end of each state-action pair transition $\langle (x, a), r, (y, b) \rangle$, the same update rule as for $V^\pi(x)$ is applied to estimate $Q^\pi(x, a)$:

$$\hat{Q}^\pi(x, a) \leftarrow \hat{Q}^\pi(x, a) + \alpha \Delta \hat{Q}^\pi, \tag{2.20}$$

where $\Delta \hat{Q}^\pi = r + \gamma \hat{Q}^\pi(y, b) - \hat{Q}^\pi(x, a)$. We notice that there is a mutual influence between the policy π and the utility function Q^π. In effect a new update of Q^π changes π, which then modifies Q^π and so on until both of them become optimal. Algorithms based on this update rule are called *Sarsa* (because of the tuple *State, Action, Reward, State, Action*) and was first investigated by Rummery and Niranjan (1994) who called it *Modified Q-learning*. Q-learning (Watkins 1989) is another algorithm also based on TD-learning, which directly estimates the optimal utility function Q^*. It uses the following update rule:

$$\hat{Q}^*(x, a) \leftarrow \hat{Q}^*(x, a) + \alpha \Delta \hat{Q}^*, \tag{2.21}$$

where

$$\Delta \hat{Q}^* = r + \gamma \max_b \hat{Q}^*(y, b) - \hat{Q}^*(x, a). \tag{2.22}$$

Unlike Sarsa, Q-learning does not need to know the actual action that will be executed during the next experience; it simply takes greedy action with respect to y and the current estimate of Q^*. Q-learning is qualified by *asynchronous* or *off-policy* algorithm because it can learn the utility function of a policy (the optimal one) while following another (by observing the behavior of another agent for instance). The convergence of these algorithms is guaranteed if all state-action pairs are visited an infinite number of time and the learning rate is decayed adequately. Moreover the Sarsa algorithm requires that the control policy converges little by little towards a greedy policy.

Eligibility Traces

One way of improving learning and dealing more efficiently with the temporal credit assignment is not only to update the value function of the state which is currently visited, but to update those that have led to it as well. To do so, we keep a record of the degree of recency of the visited states: their eligibility traces. Thus the estimate of the value function is updated for each state according to its eligibility. The update rule is

$$\hat{V}^{\pi}(x) \leftarrow \hat{V}^{\pi}(x) + \alpha\Delta\hat{V}^{\pi}e(x), \text{ for each } x \in X \tag{2.23}$$

where $e(x)$ is the eligibility of state x. It is updated on-line either by accumulating traces

$$e(x) \leftarrow \begin{cases} \gamma\lambda e(x) + 1 & \text{if } x \text{ is the current state,} \\ \gamma\lambda e(x) & \text{otherwise} \end{cases} \tag{2.24}$$

or by replacing traces

$$e(x) \leftarrow \begin{cases} 1 & \text{if } x \text{ is the current state,} \\ \gamma\lambda e(x) & \text{otherwise,} \end{cases} \tag{2.25}$$

where $0 \leq \lambda \leq 1$ is the *trace-decay factor.* The difference between these two eligibility trace mechanisms is emphasized in figure 2.5. Basically accumulating traces takes into account both the frequency and the recency of the state whereas replacing traces only considers the recency. Both traces decay exponentially according to λ when the state is no longer visited. Recent work has reported the superiority of replacing traces (Singh and Sutton 1996). Prediction algorithms based on the update 2.23 are called TD(λ) and are a

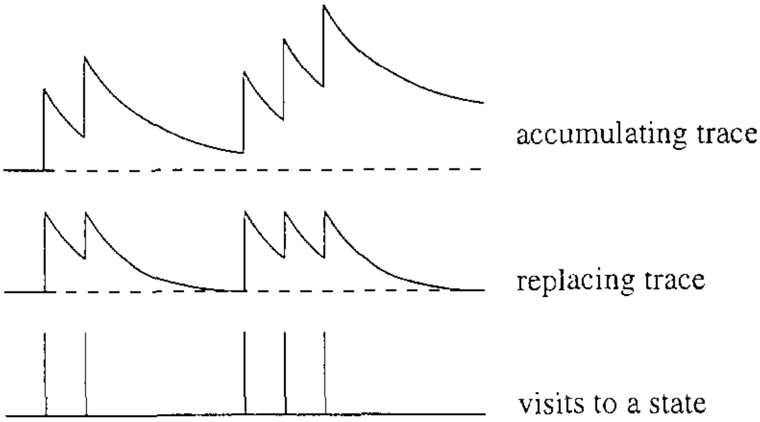

Figure 2.5: Evolution of traces according to the state visits.

generalization of TD(0). The way we introduced the eligibility traces is called *the backward view of TD(λ)* (Sutton and Barto 1998). It is intuitive and can be directly implemented.

On the other hand, *the forward view of TD(λ)* is a more theoretical view and consists in making updates using predictions on several forthcoming steps.

Eligibility traces can also be used to enhance the performances of control algorithms such as Sarsa or Q-learning. However it is required to have traces for each state-action pair and not only for each state. The algorithms resulting from this combination are Sarsa(λ) (Rummery 1995) and Q(λ) (Peng and Williams 1996), and are presented in figure 2.6.

The counterpart of the efficiency in the use of eligibility traces is their computational cost because the value function and the eligibility traces have to be updated for each state (or state-action pair for the control). However there are some promising results that overcome this drawback (Cichosz 1995; Wiering and Schmidhuber 1998). The principle is to update only the states whose traces are above a certain ϵ and ignore the remaining states.

Exploration

As it was pointed out earlier the convergence of TD control algorithms to an optimal policy is essentially subject to the requirement to visit all state-action pairs an infinite number of time. This is obviously not possible in practice because it would take too long before starting the optimal control. The agent is therefore faced with an interesting trade-off between (i) performing actions that will increase its knowledge about the environment (i.e. visiting new states or consolidating its experience) and (ii) actions that are optimal relative to its current estimate of the optimal policy. In fact some actions are known to give good results in a particular situation but some others are not known at all and might produce better results. This trade-off is called the *exploration-exploitation dilemma*. Methods to solve this dilemma can be classified into two categories: *undirected* methods and *directed* methods.

Undirected methods, also called *ad hoc* methods, do not use any knowledge about the learning process to direct the exploration: they make a random exploration. The simplest technique to do so is called ϵ-greedy policy. It takes a greedy action by default and, with probability ϵ, a random action. The parameter ϵ is set to 1 in the beginning to

$\hat{Q}(x', a') \leftarrow 0$ and $e(x', a') \leftarrow 0$ for each $x' \in X$ and $a' \in A$
Observe x
Choose a according to $\hat{Q}(x, a)$ and some exploration policy
loop
  Perform a, observe r and y
  Choose b according to $\hat{Q}(x, b)$ and some exploration policy

  For Q(λ)
  $\Delta'\hat{Q} \leftarrow r + \gamma \max_c \hat{Q}(y, c) - \hat{Q}(x, a)$
  $\Delta\hat{Q} \leftarrow r + \gamma \max_c \hat{Q}(y, c) - \max_c \hat{Q}(x, c)$

  For Sarsa(λ)
  $\Delta'\hat{Q} \leftarrow r + \gamma\hat{Q}(y, b) - \hat{Q}(x, a)$
  $\Delta\hat{Q} \leftarrow \Delta'\hat{Q}$

  for each state-action pair (x', a') **do**
    $e(x', a') \leftarrow \gamma\lambda e(x', a')$
    $Q(x', a') \leftarrow Q(x', a') + \alpha\Delta\hat{Q}e(x', a')$
  end for
  $Q(x, a) \leftarrow Q(x, a) + \alpha\Delta'\hat{Q}e(x, a)$

  For accumulating traces
  $e(x, a) \leftarrow e(x, a) + 1$

  For replacing traces
  $e(x, a) \leftarrow 1$
  for each $a' \in A$ **do**
    $e(x, a') \leftarrow 0$
  end for

  $x \leftarrow y$ and $a \leftarrow b$
end loop

Figure 2.6: Algorithms of Q(λ) and Sarsa(λ) with either replacing or accumulating traces. For $\lambda = 0$ we have Sarsa and one step Q-learning algorithms.

encourage exploration and is slowly decreased thereafter to ensure exploitation. Another more sophisticated technique is based on a Boltzmann distribution

$$P(a|x) = \frac{e^{Q(x,a)/T}}{\sum_{b\in A} e^{Q(x,b)/T}} \tag{2.26}$$

where T is the temperature parameter which controls the exploration. With a high temperature the probabilities are uniform and as T decreases the probability of choosing $\pi^*(x)$ become closer to one.

Directed methods (see (Thrun 1992; Wyatt 1997; Wilson 1996) for more details) are based on an exploration bonus which is added to the utility function. It is worth mentioning that this bonus is simply a random value in the case of indirected methods. As for directed methods, the bonus is based on one or a combination of the following criteria:

- *counter* criterion, which takes into account the number of times that a state-action pair is visited;
- *error* criterion, which uses the variation of the utility function. The higher the variation of the utility the more its corresponding state-action pair is preferred;
- *recency* criterion, which promotes state-action pairs that have been tried the least recently.

Other techniques that seem to be powerful and promising are based on the Gittins' indexes and are currently investigated by (Meuleau and Bourgine 1998).

2.3 Structural Credit Assignment

The natural and simplest way of representing the estimates of the value and utility functions is to use a lookup table. Such a table will have a single entry for each state or state-action pair. This kind of representation is well-suited for simple tasks with small state and action spaces. However when these spaces become huge, the problem faced goes beyond the prohibitive amount of memory needed to store values of each entry. Specifically, the greater the number of situations which the agent has to deal, the smaller the probability that the same situation will be faced more than once. Thus the learning process becomes difficult and the agent needs some *generalization* ability, which allows it to make a fair decision in a situation it has never faced before. This is known as the *structural credit assignment problem* and is concerned with attributing credit (or blame) to features of the faced situations in order to generalize across them.

To deal with this problem value (or utility), functions are represented using *function approximators*. An ideal function approximator should use a fixed and limited amount of resources to represent a function, have good generalization abilities and be parameterizable to allow on-line estimation of the function.

Several generalization methods and function approximators have been developed and used in reinforcement learning: methods based on Hamming distance and statistical clustering (Mahadevan and Connell 1992), *Cerebellar Model Articulation Controller* (CMAC) (Tham 1995; Santamaria et al. 1997; Benbrahim and Franklin 1997) and neural networks (Rummery 1995; Millàn 1996). Here we will focus on neural networks and on *multi-layer perceptron* (MLP) in particular because they are well-suited to implement the gradient-descent methods (a widely used method for function approximation) using the error back-propagation algorithm, and finally because it is the approximator we used in our experiments.

2.3.1 Prediction with Function Approximator

In this section we present the general algorithm that combines both temporal difference methods and function approximation techniques. It is based on the gradient-descent approach and can be used with any function approximator.

Let's assume we have at our disposal the true values of V^π (the function we want to approximate) for each $x \in X$. Also let $\hat{V}_p^\pi(x) = \hat{V}^\pi(\vec{p}, x)$ be the function which approximates V^π where $\vec{p}$ is a parameter vector. It is those parameters that are tuned so that $\hat{V}_p^\pi(x)$ becomes closer to $V^\pi(x)$ for each $x \in X$. Finding a good approximation of V^π using $\hat{V}_p^\pi$ consists in finding the configuration of $\vec{p}$ that minimizes the quadratic error over the state space:

$$E = \frac{1}{2} \sum_{x \in X} \left[V^{\pi}(x) - \hat{V}_p^{\pi}(x) \right]^2 . \tag{2.27}$$

To do so gradient-descent methods progressively reduce the observed error for each step. The parameter vector is tuned in the opposite direction of the gradient of $\hat{V}_p^{\pi}(x)$ with respect $\vec{p}$:

$$\begin{aligned} \vec{p} &\leftarrow \vec{p} - \alpha \nabla_{\vec{p}} E \\ &\leftarrow \vec{p} + \alpha \left[V^{\pi}(x) - \hat{V}_p^{\pi}(x) \right] \nabla_{\vec{p}} V_p^{\pi}(x) \end{aligned} \tag{2.28}$$

where α is the learning rate and $\nabla_{\vec{p}}$ is the gradient operator with respect to $\vec{p}$. The learning rate α weights the strength of the tuning so that only a small step is taken in the improving direction. If the learning rate is tuned to completely reduce the error on the observed example then the parameter vector will not converge because it will be destabilized after each new update.

In the case of TD learning, the value we want to approach with $\hat{V}_p^{\pi}(x)$ after an experience $< x, a, r, y >$, is $r + \gamma \hat{V}_p^{\pi}(y)$. Hence the update rules for the parameter vector are

$$\vec{p} \leftarrow \vec{p} + \alpha \Delta \hat{V}_p^{\pi} \vec{e} \tag{2.29}$$

where $\Delta \hat{V}_p^{\pi}$ is the TD error $r + \gamma \hat{V}_p^{\pi}(y) - \hat{V}_p^{\pi}(x)$, α is the learning rate and $\vec{e}$ is the eligibility trace vector. In the tabular case eligibility traces were assigned to each state. In the present case they are assigned to each component of the parameter vector. Their update is

$$\vec{e} \leftarrow \gamma \lambda \vec{e} + \nabla_{\vec{p}} V_p^{\pi}(x) \tag{2.30}$$

where $\vec{e}$ has an initial value of zero.

The equations presented here can be extended to estimate the utility function $Q^{\pi}(x, a)$ in the same way as for the tabular case. In the next section we briefly introduce neural networks, and then we show how they can be used with the above update rules.

2.3.2 Neural networks

Artificial neural networks (ANN), also known as connectionist networks, are mathematical and computational models inspired from human nervous cells. Their basic components are simple *processing units* (also called *neurons* or *perceptrons*) interconnected by *weighted synaptic links*. Each unit receives signals from other units or external sources and processes them. The result of processing is either used as input to other units or as output of the network.

Architecture

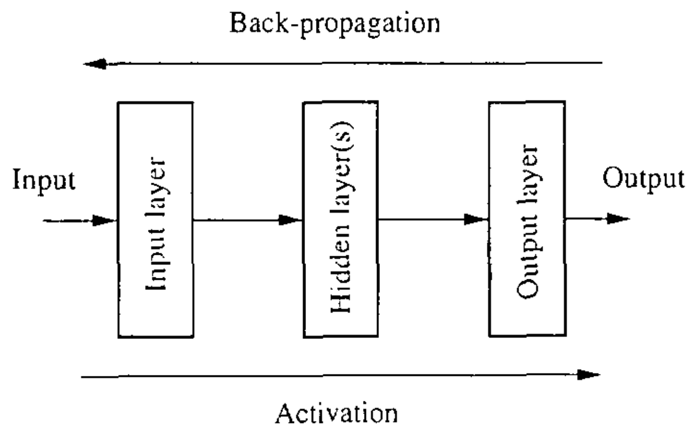

Figure 2.7: Multi-layer perceptron network.

As we said above, we will only consider *multi-layer perceptron* (MLP) networks. In such networks, units are organized in layers: units interacting with the outside are in the input or output layers, and all other units belong to the hidden layers (figure 2.7). When the units are connected in a forward way (from the input to the output layer) we have a *feed-forward network*. Sometimes certain units in the hidden or output layers are fed back to previous layers and give a *recurrent network*.

Activation

The activation in the network is computed by propagating the units activation from the input to the output. The connexion between two units is defined by a weight w_{ij}^q which determines the effect that activation a_j^{q-1} of unit j has on unit i (figure 2.8). The activation of a unit i (its output) is calculated in the following manner,

$$a_i^q = \mathcal{F}(s_i^q), \tag{2.31}$$

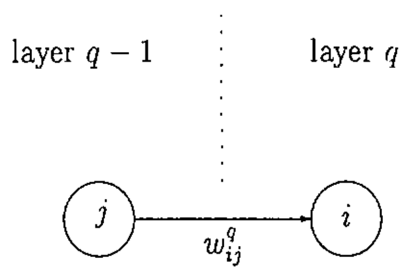

Figure 2.8: A connexion between units of consecutive layers. The index of the layers decreases from the output to the input.

where q indexes the layer, $\mathcal{F}$ is an activation function and s_i^q the weighted sum of the unit's inputs plus a bias b_i^q,

$$s_i^q = \sum_j w_{ij}^q a_j^{q-1} + b_i^q. \tag{2.32}$$

The activation function $\mathcal{F}$ has to be non-linear and is usually either sigmoidal, semi-linear or tangential. However a sigmoid activation function is very often used

$$\mathcal{F}(s) = \frac{1}{1+e^{-s}}. \tag{2.33}$$

Back-Propagation

The principle of the back-propagation method is to propagate the error, namely the difference between the desired output and the actual output, from the output to the input units so as to know the error of each unit. It consists in using a gradient-descent technique to minimize the quadratic error

$$E = \frac{1}{2}(\vec{d} - \vec{a})^2, \tag{2.34}$$

where $\vec{d}$ is the desired output vector and $\vec{a}$ is the actual output vector of the network. To do so the gradient $\partial E / \partial w_{ij}^q$ is computed by decomposing it into two terms which will be separately evaluated

$$\frac{\partial E}{\partial w_{ij}^q} = \frac{\partial E}{\partial s_i^q} \frac{\partial s_i^q}{\partial w_{ij}^q}. \tag{2.35}$$

The second term can be directly calculated

$$\frac{\partial s_i^q}{\partial w_{ij}^q} = \frac{\partial}{\partial w_{ij}^q}\left(\sum_k w_{ik}^q a_k^{q-1} + b_i^q\right) = a_j^{q-1}, \tag{2.36}$$

and the first term which is the error ε_i^q on the unit i of the layer q is decomposed once again to give:

$$\varepsilon_i^q = \frac{\partial E}{\partial s_i^q} = \frac{\partial E}{\partial a_i^q}\frac{\partial a_i^q}{\partial s_i^q}. \tag{2.37}$$

As $a_i^q = \mathcal{F}(s_i^q)$ we immediately deduce

$$\frac{\partial a_i^q}{\partial s_i^q} = \mathcal{F}'(s_i^q). \tag{2.38}$$

For the calculation of $\partial E/\partial a_i^q$ we have to consider two distinct cases in which whether layer q is or is not the output layer. If it is then

$$\frac{\partial E}{\partial a_i^q} = (d_i - a_i^q), \tag{2.39}$$

and the error of an output unit is

$$\varepsilon_i^q = (d_i - a_i^q)\mathcal{F}'(s_i^q). \tag{2.40}$$

When the layer q is not the output layer, the gradient $\partial E/\partial a_i^q$ is derived from the errors of forward layers

$$\begin{aligned}\frac{\partial E}{\partial a_i^q} &= \sum_k \frac{\partial E}{\partial s_k^{q+1}}\frac{\partial s_k^{q+1}}{\partial a_i^q}\\ &= \sum_k \varepsilon_k^{q+1} w_{ki}^{q+1}.\end{aligned} \tag{2.41}$$

and the error of an non-output unit is

$$\varepsilon_i^q = \left(\sum_k \varepsilon_k^{q+1} w_{ki}^{q+1}\right)\mathcal{F}'(s_i^q). \tag{2.42}$$

Finally each weight of the synaptic links is corrected as follows

$$w_{ij}^q \leftarrow w_{ij}^q + \alpha\varepsilon_i^q a_i^q, \tag{2.43}$$

where ε_i^q corresponds either to the one of equation 2.40 or 2.42. At this stage it is straightforward to notice how the gradient-descent method for value function prediction presented in section 2.3.1 can be easily implemented with neural networks. Figure 2.9 presents the connectionist version of Sarsa(λ).

Initialize $\vec{w}$ with small random values and $\vec{e}$ to zero
Observe x
Choose a according to $\hat{Q}_w(x, a)$ and some exploration policy
loop
  Perform a, observe r and y
  Choose b according to $\hat{Q}_w(y, b)$ and some exploration policy

  $\Delta\hat{Q}_w \leftarrow r + \gamma\hat{Q}_w(y, b) - \hat{Q}_w(x, a)$
  $\vec{e} \leftarrow \gamma\lambda\vec{e} + \nabla_{\vec{w}}\hat{Q}_w(x, a)$
  $\vec{w} \leftarrow \vec{w} + \alpha\Delta\hat{Q}_w\vec{e}$

  $x \leftarrow y$ and $a \leftarrow b$
end loop

Figure 2.9: Algorithm of Sarsa(λ) with a connectionist function approximator.

2.3.3 Connectionist Reinforcement Learning

To represent the utility with MLP networks (called in this case *Q-nets*), one has to carefully define a certain number of issues.

Basically Q-nets take as inputs a state x and an action a and produce their utility $Q(x, a)$ as an output. So the first issue concerns the use of a single network whose inputs encode both the state and the action or a set of $|A|$ distinct networks whose inputs encode only the state. The monolithic case may give fair results when the action space is continuous but is not efficient to deal with domains with discrete actions. This limitation comes from the fact that the network has, in this case, to model a highly non-linear function because for the same state different actions (usually having a similar representation) may have very different utility functions. Moreover this architecture does not support the use of eligibility traces. The distributed architecture, also called OAON (*One Action One Networks*) (Lin 1992) associates one network to each action to reduce the interferences between actions and is suitable for use with eligibility traces.

The second issue concerns non-Markov states. Recall that a Markov state is necessary and sufficient to make the right decision and to predict the next state for a given action in a given state. When the agent does not have a Markov state it faces the *hidden state* *problem*. To cope with this problem the agent has to build an internal Markov state using history information. Recurrent neural networks, construct such a history in a compact way: units in the hidden layer are fed back to a part of the input layer called *context*, the rest of the input layer is devoted to the state (figure 2.10). These networks are known as Elman networks and have been used by Lin (1992) to solve several non-Markov tasks.

The last issue refers to the specifications of each of the three layers.

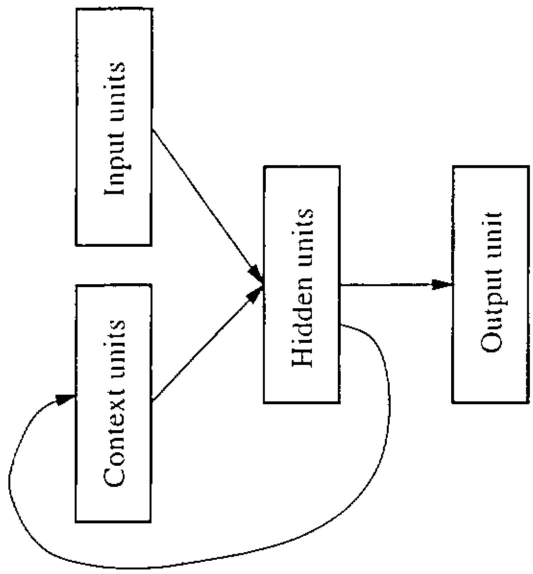

Figure 2.10: An Elman network as used by Lin (1992).

The Input Pattern

The input vector of the neural network is a representation in terms of *features coding* of the state to be evaluated. It is called the input pattern. The design of this vector is very important and has a great impact on the learning and generalization abilities of the network. The choice of the features requires a good knowledge of the task domain and their coding depends on their nature.

As far as the features allow it, the most simple and efficient way of representing them is a binary coding. If a feature has a finite and small number of possible values, for instance such as a lift's location in a building, then one input unit is associated with each of them. The unit is 'on' when the feature has the corresponding value and 'off' otherwise.

When the feature is a real value, such as a robot's sensor reading, it can be either scaled in the range [0, 1] (to avoid units overshooting) and represented with a single unit, or spread over several units. The latter choice is a *coarse coding* technique and is useful when different responses are needed for different ranges of the value we want to code or also when we need more accuracy. The coarse coding technique is used in conjunction with binary features, radial basis functions (RBF) or sigmoid functions. For more details about these techniques see (Sutton and Barto 1998) for the first two methods and (Rummery 1995) for the third one[1].

The Hidden Layer(s)

The number of hidden layers as well as the number of units in each layer are the factors that define the degree of freedom of a neural network. Hence the more complicated the function, the more numerous hidden layers and units. In an MLP a single hidden layer is usually sufficient but there is no systematic means of determining the exact number of hidden units. However it has been reported by Rummery (1995) that, in reinforcement learning applications, the final performance of the system is no more affected beyond a certain number of hidden units. Only the convergence time and the computational cost become high. Therefore a possible strategy to find the ideal number of hidden units would be to start with a small number of hidden units and to increase it up to the point where no improvement can be observed.

The Output Layer

The output of the network, when it is used to approximate a utility function, is a real value. It can be either encoded by several sigmoidal output units using the technique of overlapping Gaussian ranges (Pomerleau 1991) or by a single unit. In the latter the activation function of this unit may be either linear or sigmoidal. However with a linear function the output value is not bounded, therefore a high error may be back-propagated and thereby makes the units overshot. If a sigmoid function is utilized, the output value is

[1] A brief description of the coarse coding technique using sigmoid function is given in section 4.4.2.

within the range [0, 1] so the immediate reinforcement has also to be within this range. In practice either we have an idea about the variation range of the reinforcement, so we can scale it, or we use a very small learning rate which will slow-down the learning process. To overcome this handicap Benbrahim and Franklin (1997) developed a method called Self Scaling Reinforcement (SSR) which self scales the reinforcement signal according to the minima and the maxima observed.

2.4 Summary

This chapter has set up the foundations of reinforcement learning and has overviewed related existing methods and algorithms. Let's recall that reinforcement learning has to be viewed as a class of problems or as an adaptive control paradigm rather than a particular learning technique. RL has become very popular in the field of intelligent autonomous agents and has attracted researchers from other disciplines like statistics, psychology and artificial intelligence. RL is becoming increasingly mature because, on one hand its theoretical aspects (link with dynamic programming, choice of optimality criteria, analysis of various algorithms' behavior, function approximators) are intensively investigated and on the other hand the number of practical applications is continuously growing. Examples of such applications are: elevator control (Crites 1996), TD-Gammon (Tesauro 1995), dynamic channel allocation in cellular telephone system (Singh and Bertsekas 1997) and job-shop scheduling (Zhang and Dietterich 1995). The efforts are currently focused on scaling up reinforcement learning to large, complex and partially observable problems. They involve issues such as continuous state and action spaces, representation, hierarchical control and task decomposition, and methodologies for general application of RL. The last two issues constitute the central theme of this thesis.

Chapter 3

The Postman Robot Problem

In this thesis the postman robot problem is used, as an application framework for the methodology we will introduce, and as a testbed for our experiments. In this chapter we describe the postman robot task as well as the robot and the particular setups that we have used.

3.1 The Postman Robot Task

The postman-robot is given a set of parallel and conflicting objectives and must satisfy them as best as it can. The robot acts in an office environment composed of offices, a batteries' charger and a mailbox. Its task is to collect letters from the offices and post them in the mailbox. While achieving its postman's task as efficiently as possible the robot has to avoid collisions with obstacles and recharge its batteries to prevent break-downs.

3.2 The robot

The physical robot is a Nomad 200 mobile platform (figure 3.1). It has 16 infrared sensors for ranges less than 40 centimeters, 16 sonar sensors for ranges between 40 and 650 centimeters, and 20 tactile sensors to detect contact with objects. It is also equipped with wheel encoders and a compass to compute its current location and orientation relative to its initial ones. Finally, it has three wheels controlled together by two motors which make it translate and rotate. A third motor controls the turret rotation.

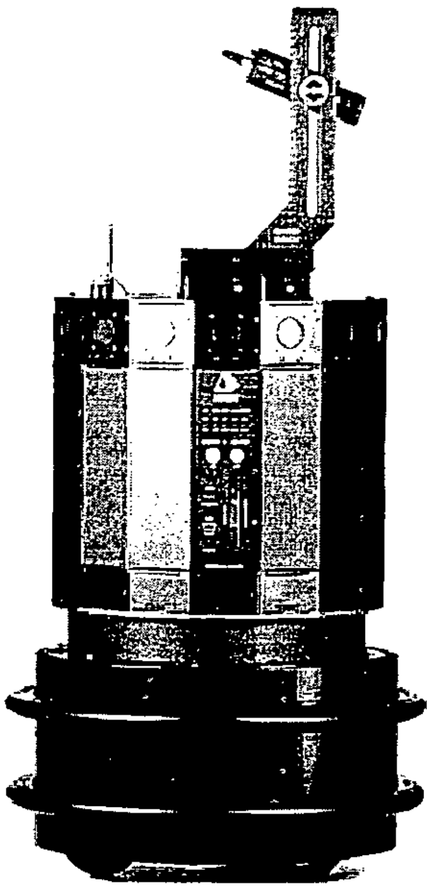

Figure 3.1: The Nomad 200 robot

3.3 The Environment

The postman robot's decisions are mainly driven by the letters flow, as well as by the batteries' level. In this section we describe their dynamics and relative assumptions.

3.3.1 Assumptions

We define an *atomic action* that the robot can perform as a steering of $\Delta\theta$ degree followed by a translation of Δd centimeters. Thus the set of available actions is constituted of several pairs $a_i = (\Delta\theta_i, \Delta d_i)$. The interval between the end of the execution of two actions defines the duration of an interaction cycle and corresponds to one time step.

In addition the following assumptions are made about the robot capabilities:

- The robot can sense the number of letters it holds, its batteries' level and the number of letters in each office;
- The robot gets the letters once it is in an office, posts the letters once it is near the mailbox, and recharges its batteries once it is near the charger (because it does not have any grasping or recharging devices).

3.3.2 Dynamics

Let's denote $x_r(t)$ the number of letters that the robot holds, $x_{li}(t)$ the number of letters in each office i, and $x_b(t)$ the batteries' level, at a given time step t. The evolution of these parameters are governed by the following equations:

- Letters in an office i

$$x_{li}(t+1) = \begin{cases} x_{li}(t) + \xi(t) \\ 0, \text{if the robot picks up the letters from office } i \end{cases}$$

  where $\xi(t)$ is the incoming letters in office i at time step t.

- Letters transported by the robot:

$$x_r(t+1) = \begin{cases} x_r(t) + x_{li}(t), \text{if the robot picks up the letters from office } i \\ 0, \text{if the robot posts the letters it holds.} \end{cases}$$

- Batteries' level:

$$x_b(t+1) = \begin{cases} x_b(t) - x_{bc} \\ \\ 100\%, \text{if the robot recharges its batteries} \end{cases}$$

where x_{bc} is the batteries' consumption rate for one time step.

3.3.3 Testbed

The particular environment we used for our experiments is composed of three offices, one mailbox and one charger (figure 3.2). Its size is approximatively $13m \times 13m$. Letter arrivals in each office are either periodic (i.e n letters each p time steps) or follow a Poisson distribution. Table 3.1 shows the letters flow patterns that were used.

	Periodic (letters/period)	Poisson (mean letters/time steps)
Office 1	1/40	3/100
Office 2	1/30	5/100
Office 3	1/20	7/100

Table 3.1: The letter arrivals patterns for each office.

To carry out the experiments we had at our disposal the Nomad 200 development host which simulates the robot's sensors and cinematics and we wrote a program which simulates the letters arrival and the batteries' dynamics (figure 3.2). Although the robot's simulator is realistic it is time consuming. For example, it takes about 30 seconds to move from one office to another when the simulator is run on a Sun Ultra 1 station. To speed up the simulation process, we have proceeded in the following manner. When the navigation's behaviors were learned (using the Nomad 200 simulator) we measured the number of time steps needed to move from one place to another. These measures are used to define a grid simulator which is then coupled to the letters flow and batteries simulator.

Thus we can test and validate the coordination of these elementary behaviors much faster, while still being able to reuse the learned coordination with the robot's simulator.

As our navigation algorithms rely on the odometry (see section 4.5.1), we were unable to reuse them on the real robot because of the drift. We are currently developing other navigation behaviors based on beacons' detection.

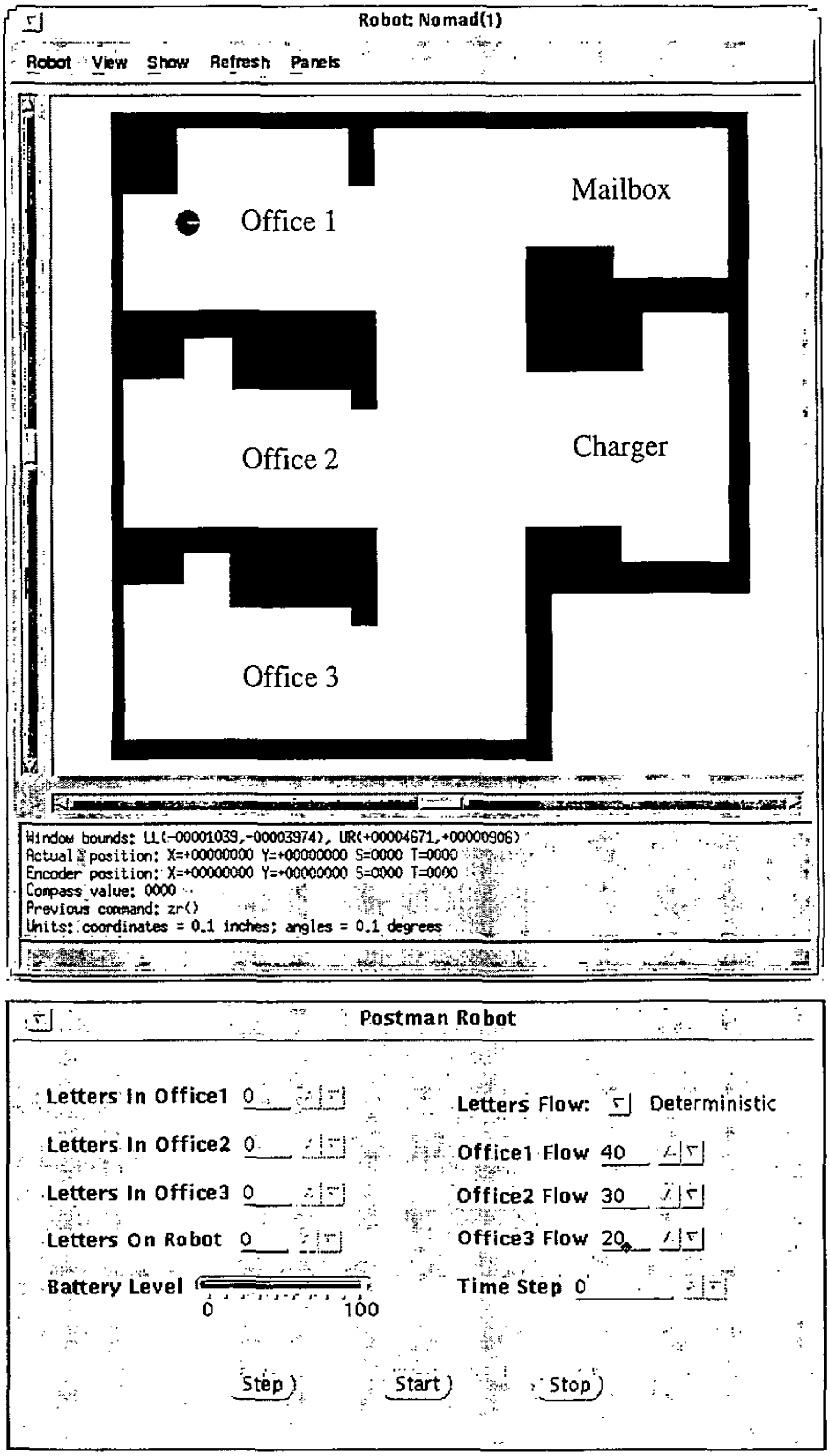

Figure 3.2: The Nomad 200 development host and the letters flow and batteries simulator.

3.4 Summary

We have chosen the postman robot task because it provides an opportunity to apply reinforcement learning to build both reactive (navigation, obstacle avoidance) and planning (collecting and posting letters efficiently) skills of the robot. It is an instance of a more general task involving the coordination of concurrent and interfering behaviors and is analogous to the optimal foraging problem which is usually faced by animals (Stephens and Krebs 1986). Let's add by the way that a postman robot is currently running in a building of Carnegie Mellon University and that its design and implementation has involved about 10 persons (Simmons et al. 1997).

Chapter 4

The Methodology

This chapter introduces a methodology to solve problems using reinforcement learning. We begin this chapter by justifying the need of a methodology in reinforcement learning. An interaction model between the agent and the environment is then presented and some important notions like agent or behavior are clarified. Then we describe the *Hierarchical Problem Solving* (HPS) methodology as well as its associated methods, and apply it to the postman robot problem. Finally we report the experiments we carried out and the results we obtained.

4.1 A Methodology for Reinforcement Learning

Problem solving using embedded reinforcement learning agents has become very attractive because the level of abstraction at which the designer intervenes is raised (i.e. the agent is told what to do using the reinforcement function and not how to do it) and little programming effort is required (most of the work is done by autonomous training).

Nevertheless and despite its mathematical foundations, reinforcement learning cannot be used as it is to make the agents solve complex problems. Such a limitation is essentially due to the huge search space the agent has to deal with and to the difficulty in finding the adequate reinforcement function. One way to solve complex problems is to adopt a divide-and-conquer approach: (1) breaking down the initial problem into sub-problems with small state spaces and simple reinforcement functions, (2) solving each sub-problem, (3) combining the solutions of each sub-problem to solve the original problem.

The above procedure is recognized to give fair results and has been widely applied in reinforcement learning (see (Mahadevan and Connell 1992; Lin 1993; Kalmàr et al. 1998; Dietterich 1997) for instance). However only experienced designers can overcome the tricks that may appear during its use. In this chapter we introduce a methodology which integrates this procedure and helps the designer to build efficient control architectures for reinforcement learning agents.

The objective of a methodology, in any engineering field, is to provide helpful guidelines to engineers during the design process. Its role is of great importance because it not only ensures the quality of the final product but also optimizes the use of available resources, the tasks' allocation over several persons as well as the management of the whole process. The different stages in a general engineering methodology are shown in figure 4.1. The next two sections review attempts to determine principles for the agent's design process.

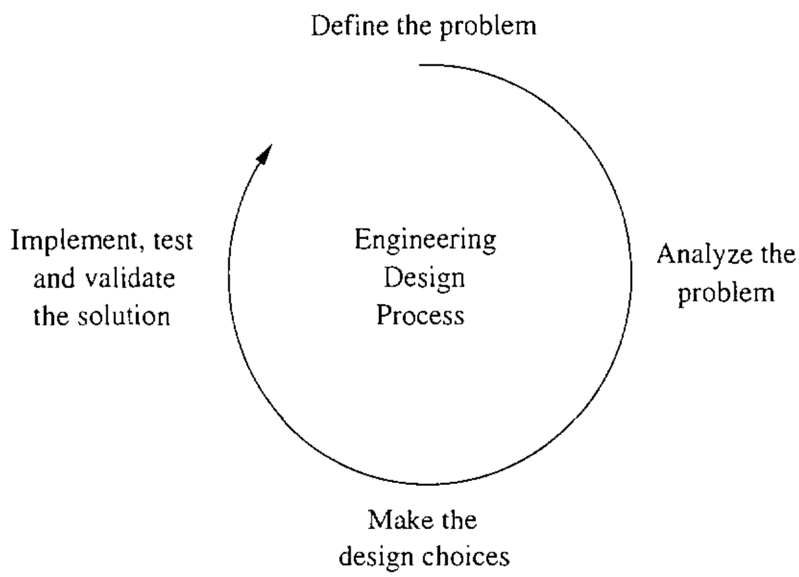

Figure 4.1: A general engineering methodology.

4.1.1 Pfeifer's Design Principles

By setting up the foundations for autonomous agents' design principles, Pfeifer (1996) wanted to provide new insights in understanding intelligence. His main argument is that the best way to understand intelligence is to build autonomous agents. Another major purpose is that the agent's design relies on the intuitions of experienced designers and that this know-how is often left implicit in most scientific publications. Thus the design principles aim at making this knowledge explicit and provide guidance on how to build autonomous agents.

The design principles which were proposed are clustered into two classes. The first class is called *task environment* and concerns the definition of the ecological niche in which the agent will evolve, as well as the task it has to achieve and the behaviors it has to exhibit. The second class is devoted to the design of the agent itself and is constituted of seven principles which include issues such as agent morphology and control architecture. We review these principles as they were summarized in (Pfeifer and Scheier 1998):

1. **The complete agent principle.** The kind of agents of interest are the complete agents, i.e. agents that are autonomous, self-sufficient, embodied and situated.

2. **The principle of parallel, loosely coupled processes.** Intelligence is emergent from an agent-environment interaction based on a large number of coupled processes that run in parallel, loosely coupled processes that run asynchronously and are connected to the sensory-motor apparatus.

3. **The principle of sensory-motor coordination.** All intelligent behavior (e.g. perception, categorization, memory) is to be conceived as a sensory-motor coordination which serves to structure the input.

4. **The principle of cheap designs.** Designs must be parsimonious, and exploit the physics and the constraints of the ecological niche.

5. **The redundancy principle.** Sensory systems must be designed based on different sensory channels with information overlap.

6. **The principle of ecological balance.** The "complexity" of the agent has to match the complexity of the task environment. In particular given a certain task environment, there has to be a match in the complexity of sensors, motor system, and neural substrate.

7. **The value principle.** This principle states that the agent has to be equipped with a value system and a mechanisms for self-supervised learning employing principles of self-organization.

These design principles were successfully applied to build *"Sahabot"*, a mobile robot whose behavior is inspired from the desert ant's behavior.

4.1.2 The BAT Methodology

The need for a principled approach to developing learning autonomous agents, also motivates the efforts of Dorigo and Colombetti (1998) to define a new technological discipline called *Behavior Engineering*. Behavior Engineering aims at providing a methodology, a repertoire of models and a set of tools supporting all the phases of the agent development process. The methodology they proposed, called *Behavior Analysis and training* (BAT) (Colombetti et al. 1996), is based on the experience acquired during their past research, and covers several issues in the building process of autonomous robots such as specification, design, training, and assessment. The BAT methodology comprises the following stages:

1. The informal (i.e. in natural language) description of the agent and its environment as well as the requirements of the desired behavior.

2. The analysis of the behavior and its decomposition into simple behaviors. The interaction between these behaviors is then defined using some operators (independent sum, combination, suppression, sequence). The result of this stage is a structured behavior.

3. The specification of the robot components including in particular the sensors and the effectors, the controller architecture, the reinforcement function for each elementary behavior, the training strategy, and sometimes the extensions that should be added to the environment. A set of generic control architectures based on *Behavioral Modules* (BM) is provided to implement the structured behavior.

4. The design, the implementation and the verification of the control architecture.

5. The robot's training until the desired behavior is learned.

6. The validation of the learning process and the observed behavior.

This methodology assumes that the robot's apparatus and the environment are pre-defined, and the BMs are endowed with a well-chosen reinforcement learning mechanism which was in most cases a *Learning Classifier System* (LCS). The feasibility of this methodology was demonstrated through three practical examples.

4.1.3 Discussion

The two approaches presented above constitute the main and, to the best of our knowledge, the only attempts to define a principled and systematic means to designing autonomous agents. Both of them were developed within and especially for the robotics field. However some remarks can me made about them.

Pfeifers' design principles provide a set of recommendations and advice to respect, rather than guidelines to follow. Also they do not deal with the testing and the evaluation issues, and timidly address the learning aspect. However the difference between a behavior and the mechanism which produces it by interaction with the environment has been clearly stated and highlighted (this point will be detailed in the next section).

The BAT methodology explicitly guides the designer during all the stages and defines the expected result at the end of each of them. Learning is considered as an integrated part of the methodology and the role of the trainer to make the learning process efficient is stressed. However we regret a certain lack of formalism in the specification phase and that the decomposition process heavily relies on the designer's intuition and past experience.

In conclusion, we can point out that these approaches are (or may be) complementary in the sense that the first one addresses the scientific part while the second one addresses the engineering part in the design of autonomous agents.

4.2 Agent-Environment Interaction Model

At this stage it is worth clarifying the notion of *behavior* which is usually encountered in agent applications and robotics in particular. A behavior is the description from the observer's point of view, at different levels of abstraction, of a sequence of actions produced by the agent via its coupling with the environment. In simple words, an agent's behavior can also be defined as the result of the interaction between the agent's sensory-motor loops and the environment. In this section, we describe this interaction within the reinforcement

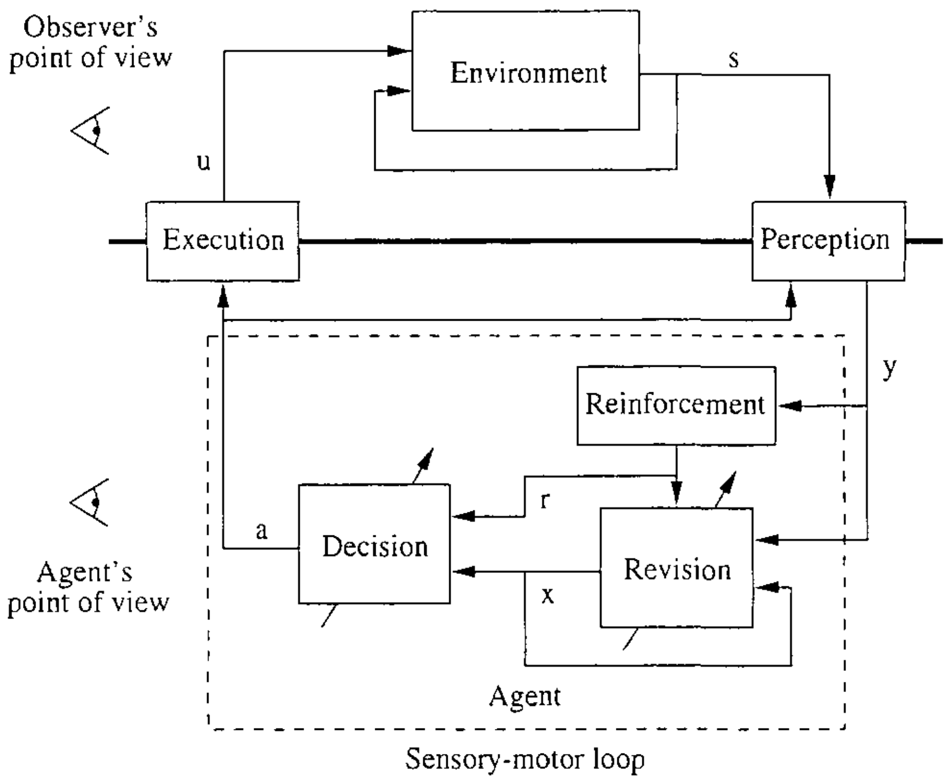

Figure 4.2: Agent-environment interaction model in a reinforcement learning framework.

learning framework, in more depth than in chapter 2. As shown in figure 4.2 the agent's behavior is modeled as a coupling of two dynamical systems: the agent, constituted here by a single sensory-motor loop and the environment. We also distinguish between the different points of view:

- the agent's point of view, which takes into account the internal mechanism that generates commands according to perceptions;

- the external observer's point of view, which considers the environment, including the agent because it is embedded in it.

This distinction allows us to emphasize the following points:

- the difference between the environment's state s and the agent's perception y, as well as between the command a that the agent executes and the action u that actually influences the environment. In robotics, for example, the agent may have an obstacle in front of it and only gets sonar or infrared readings. The observer knows that this measure is correlated with the distance to the obstacle but *a priori* not the agent. In the same context, the agent may send the motors a command corresponding to a certain number of wheel turns which makes it move in the environment, but this movement is not perceived as such by the agent. Moreover the same number of wheel turns may result in different movements according to the distance to the obstacle and to possible slipping;

- the agent's decision is taken according to the internal state x, which has to be Markov. This state is made Markov by the revision (or reconstruction) process which ranges from the identity function up to the most sophisticated knowledge revision process;

- the reinforcement signal which previously came from the task (figure 2.1) is now a part of the agent. More precisely, it is a part of the agent's *a priori* knowledge given by the designer: the *philogenetical inheritance.*

- complex behaviors may be produced by simple mechanisms through their interaction with the environment (Braitenberg 1984; Pfeifer and Scheier 1998). Hence the behavior's design process would be a projection from the problem's *domain* (observer's point of view) to the *co-domain* (robot's point of view).

From now on, we will use the term *behavior* to describe an agent solving a problem. Also, *problem decomposition* and *sub-problem* will be replaced by *behavior decomposition* and *sub-behavior.* Thus a behavior is constituted by a hierarchy of sub-behaviors just as if we had a hierarchy of agents in which each agent is solving a sub-problem. In addition, these changes will place stress on the design of an interaction rather than that of an isolated agent.

4.3 The HPS Methodology

The *Hierarchical Problem Solving* (HPS) methodology we propose aims at providing a systematic approach in the use of embedded reinforcement learning agents to solve problems. It focuses on the agent's design and more specifically on the hierarchical aspect of the control architecture. The methodology assumes that the environment, the agent and its interaction devices, as well as the problem to solve are predefined.

The HPS methodology will guide the designer by telling him how to:

- formally specify the agent's behavior;
- decompose the global behavior into a hierarchy of sub-behaviors;
- produce elementary behaviors of the hierarchy, i.e. behaviors of the lowest level, using reinforcement learning sensory-motor loops;
- coordinate the sensory-motor loops at a given level of the hierarchy to get the behavior of the upper level;
- evaluate and validate the global behavior.

Figure 4.3 gives an overview of the different stages of the methodology. We notice that:

- the controller's design is iterative, that is, the results of the global behavior's evaluation can be used to correct the specifications. The cycle is repeated until the expected behavior is observed;
- the analysis process is top-down and from the observer's point of view while the design process is bottom-up and from the robot's point of view;

- the distinction between the different points of view allows us to identify which parts have to be treated by the designer and which have to be learned by the robot. Hence we can easily combine engineering and evolution.

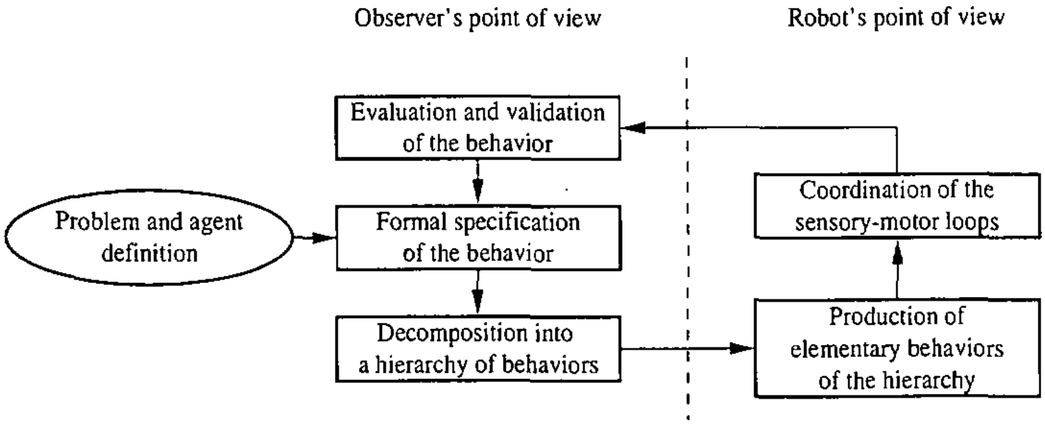

Figure 4.3: Overview of the HPS methodology

4.3.1 Specification

The specification stage has an important role in the HPS methodology. On one hand all the next stages rely on it, and on the other hand it provides the assessment stage with a useful reference matching. The dynamics of the interaction between the agent and the environment was formalized as an MDP. Thus a behavior will be represented by a particular trajectory in the MDP's state space.

By associating with each possible trajectory, a *quality criterion* we then have a means of specifying the desired behavior. The quality criterion can be expressed as the combination of an *objective function* and some *constraints* on the trajectory. The objective function largely depends on the nature of the problem and represents a measure of the system's performance such as the letters collected or the fuel consumption or more generally the squared deviation from an optimal value. It is expressed as an integral on the trajectory generated by a control policy π, for a horizon N:

$$J(\pi) = \int_0^N f(x(t), t)\, dt. \tag{4.1}$$

The constraints set $C = \{x \in X \mid \varphi_1(x) = 0, ..., \varphi_n(x) = 0\}$ reflects the aspects of the trajectory which are undesirable. So the goal is to optimize the objective function while at the same time satisfying the constraints. The constraints are enforced by augmenting the objective function as follows

$$\begin{aligned} J'(\pi, \lambda) &= J(\pi) + \int_0^N \sum_i \lambda_i(x(t), t).\varphi_i(x(t), t)\, dt \\ &= \int_0^N [f(x(t), t) + \sum_i \lambda_i(x(t), t).\varphi_i(x(t), t)]\, dt \\ &= \int_0^N F(x(t), \lambda, t)\, dt \end{aligned} \tag{4.2}$$

where the auxiliary function $F(x, \lambda)$ is called *Hamiltonian function* and λ_i are the Lagrange multipliers. They are computed using the exterior penalties method (Minoux 1986): $\lambda_i(x) = 0$ if the constraint $\varphi_i(x) = 0$ is satisfied and $\lambda_i(x) = \lambda_{pi}$ otherwise. The positive constant λ_{pi} weights the strength of the penalty.

At the end of this stage the desired behavior is specified.

4.3.2 Decomposition

Human designers are usually skillful in decomposing a complex task. However with a systematic approach they can perform better decompositions.

To decompose the main behavior into a hierarchy of sub-behaviors we propose a graphical based approach. The first step in the decomposition procedure is to graphically represent F as a function of time steps or decision steps. The next step consists in identifying the positive contributions of the agent to optimize this function as well as the associated decision making (or behaviors selection). These contributions usually appear as falling edges in the case of a minimization. Of course between two falling edges other decisions could have been made except that they do not have a positive contribution or their contribution does not appear because of the nature of the function and the kind of representation.

The surface corresponding to the integral we have to minimize is decomposed into a series of rectangles whose sides are respectively the distance between two falling edges and the value of the function when the second falling edge occurs (figure 4.4). We notice that the sum of the rectangles' surface is not exactly equal to the integral of F but to the actual measure of the agent's contribution. This measure concerns aspects of the environment that are controllable by the agent and allows us to compare the performances of two agents. For example, in the postman robot problem, the robot can choose in which office to go but cannot act on the letters' flow. In effect, while the robot is moving towards a given place, the number of letters in standby in the offices as well as the batteries' level evolve independently of the robot destination. They are actually affected when the destination is reached, that is, when the execution of the robot decision is completed. The surface of each rectangle can be minimized by reducing one of its two sides. The processes consisting in minimizing each of these sides correspond to two concurrent behaviors.

The obtained behaviors are then formally specified and decomposed once again. The procedure is repeated until the behaviors cannot be decomposed anymore or can easily be produced. At that time we have a hierarchy of sub-behaviors.

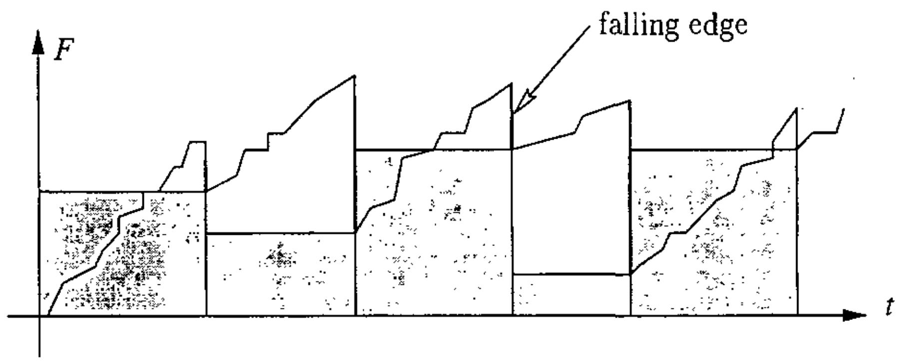

Figure 4.4: A graphical representation of the function to optimize.

Mathematical Support

In this section we provide a mathematical support for the graphical-based decomposition method presented above. Let's first introduce the fundamental definition and theorem (taken from (Minoux 1986)) on which function decomposition methods rely.

Definition We say that a function f is *decomposable* into f_1 and f_2 if f is separable (i.e. it can be put into the form $f(x, y) = f_1(x, f_2(y))$), and if moreover the function f_1

is *monotone non-decreasing* relative to its second argument. The following fundamental result can then be stated:

Theorem Let f be a real function of x and of $y = (y_1, ..., y_k)$. If f is *decomposable* with $f(x, y) = f_1(x, f_2(y))$ then we have

$$\min_{(x,y)} f(x, y) = \min_x\{f_1(x, \min_y\{f_2(y)\})\}$$

The minimization of a rectangle surface $S = l_1.l_2$ can then be written:

$$\min_{(l_1,l_2)} l_1.l_2 = \min_{l_1} f_1(l_1, \min_{l_2} f_2(l_2)),$$

where $f_1(u, v) = u.v$ and $f_2(x) = x$, when l_1 and l_2 are both positive.

4.3.3 Sensory-motor Loop's Design

In this section we present a generic sensory-motor loop (figure 4.5) which allows us to generate a behavior given its specifications. This stage of the methodology essentially consists in making design choices and concerns elementary behaviors as well as other sub-behaviors of the hierarchy.

The core of the sensory-motor loop is the learning system which computes the utility of each command. The nature of the representation of the utility function depends on the size of the state space. A simple lookup table is sufficient for small spaces, but a function approximator such as those presented in section 2.3 is needed for huge spaces.

From the perceptions we have to generate an internal state representation which must be on the one hand complete enough to allow prediction of future states and rewards and on the other hand selective, i.e. containing only information which is relevant to the behavior associated to the sensory-motor loop. Such a representation can also be learned, as reported by McCallum (1996).

The reinforcement function is an important part of the sensory-motor loop and a great care must be taken to ensure that it will lead to the desired behavior. It translates the agent perceptions' into a reward value.

The different exploration strategies were presented in section 2.2.3 so the designer can choose the most suitable among them.

Finally, as an output, the sensory-motor loop generates signals which activate or inhibit the commands. The command set may contain atomic commands which directly interact with the environment or sensory-motor loops in the case of a coordination.

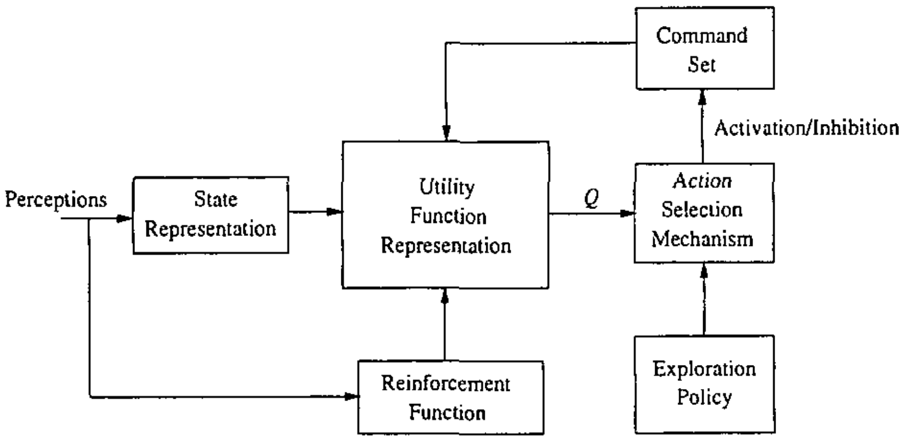

Figure 4.5: The proposed generic sensory-motor loop.

The Reinforcement function

To avoid the generation of wrong behaviors we propose to use the function that specifies the behavior to define the reinforcement function. As the specification function is defined from the observer's point of view, the expected behavior will be generated when this function is optimized. We then define the instantaneous reinforcement as the difference between the surfaces of two consecutive rectangles:

$$r(T) = F(x(T-1), \lambda).\Delta_{T-1} - F(x(T), \lambda).\Delta_T \tag{4.3}$$

where T is a decision step and Δ_T is the difference, in terms of time steps, between two decision steps $T-1$ and T. The reinforcement function has the form of a gradient and gives a continuous information on the progress made by the agent. In addition, the learning is speeded up and the exploration is improved (Mataric 1994). Given that the reinforcement learning algorithms we use maximize the cumulated discounted reward over an infinite horizon, we have:

$$
\begin{aligned}
\sum_{T=0}^{\infty} \gamma^T r(T+1) &= \gamma^0 (F(x(0),\lambda).\Delta_0 - F(x(1),\lambda).\Delta_1) \\
&\quad + \gamma^1 (F(x(1),\lambda).\Delta_1 - F(x(2),\lambda).\Delta_2) \\
&\quad + \dots \\
&\quad + \gamma^n (F(x(n),\lambda).\Delta_n - F(x(n+1),\lambda).\Delta_{n+1}) \\
&\quad + \dots \\
&= \gamma^0 F(x(0),\lambda).\Delta_0 \\
&\quad + (\gamma^1 - \gamma^0) F(x(1),\lambda).\Delta_1 \\
&\quad + (\gamma^2 - \gamma^1) F(x(2),\lambda).\Delta_2 \\
&\quad + \dots \\
&\quad + (\gamma^n - \gamma^{n-1}) F(x(n),\lambda).\Delta_n \\
&\quad + \dots \\
&= \gamma^0 F(x(0),\lambda).\Delta_0 \\
&\quad + (\gamma - 1) \sum_{T=1}^{\infty} \gamma^T F(x(T),\lambda).\Delta_T \\
&= (\gamma - 1) \sum_{T=0}^{\infty} \gamma^T F(x(T),\lambda).\Delta_T + F(x(0),\lambda).\Delta_0
\end{aligned} \tag{4.4}
$$

We notice that maximizing equation 4.4 is equivalent to the initial objective which is to minimize equation 4.2, because $0 < \gamma < 1$ and as far as the value of γ is chosen so that γ^N become negligible.

The Internal State

To build an internal state that meets the completeness and selectiveness properties, the designer has to consider the following two guidelines. First he has to identify the perceptions on which the specification function depends. that is, those which make the function change when they evolve. Then the designer has to check if the instantaneous perceptions are sufficient to make efficient decisions. If not, some kind of context or short term memory has to be added.

4.3.4 Coordination

The next chapter is entirely devoted to this stage of the methodology.

4.3.5 Evaluation and validation

During this stage the designer has to answer the following questions:

- Is the observed behavior correct ?
- If not, why ?
- What are the agent performances ?

Wyatt et al. (1998) argue that a correct approach is to employ multiple forms of evaluations. Thus it is possible to disambiguate the error source and to provide explanations of why the agent failed or succeeded.

Here we make the distinction between the behavior assessment (Colombetti et al. 1996) and the evaluation of the agent learning. The former is a *qualitative* criterion and the latter is a *quantitative* criterion. Moreover we add two viewpoints: *internal* and *external*.

To assess a behavior the designer should validate its *correctness* and its *robustness*. This is done from the observer's point of view. A behavior is correct when the task assigned to the agent is fulfilled. For example, we will validate the postman-robot if we see the robot collecting and posting the letters without running out of energy. On the other hand a behavior is robust if it remains correct when structural changes of the environment occur. Robustness is strongly linked to the adaptiveness property. If the correct behavior is not generated, then the designer should verify the learning system qualitatively, that is, determine if the agent is learning or not. A problem during this verification is usually due to a programming error in the software architecture.

	Qualitative	Quantitative
Internal	Is the robot learning ?	Convergence speed Average reward
External	Correctness Robustness	Objective function Constraints violation Failure or success rates

Table 4.1: Outline of the evaluation forms.

If the agent is effectively learning, then it is necessary to check if it is learning correctly with regards to the reinforcement program, i.e. maximizing rewards and minimizing punishments. This is done from the agent's point of view. The average of the rewards received over time steps is a good indicator to use during this checking. This is a nice way to find out why the behavior is incorrect. In effect if the agent learns what it is taught (through the reinforcement program) and exhibits the wrong behavior then it is surely because it is learning from an incorrect reinforcement function. Therefore the designer has to correct it.

Finally it is useful to compare the performances of several agents, architectures or algorithms. It is possible to evaluate the asymptotic convergence to the optimal behavior (Kaelbling et al. 1996) regarding two quantitative criteria. The first criterion is the convergence speed, that is, the necessary time (number of interaction cycles) to reach a plateau. The second criterion is the quality of the convergence. It is represented by the value of the reached plateau. The metrics that are usually used for such a comparison are cumulated deviations from the optimal behavior (if it is known), average reinforcements received over time and success or failure rates. Figure 4.1 outlines the different forms of evaluation.

4.4 Case Study

In this study we describe the application of the HPS methodology to solve the postman robot problem.

4.4.1 Specification and Decomposition

To fulfill its task, the *postman* robot has to minimize the number of letters x_{li} in the offices as well as the number of letters x_r it holds by respectively collecting and then posting them. The following objective function is derived

$$f_1(x,t) = \sum_i x_{li}(t) + \beta x_r(t), \qquad 0 < \beta < 1 \tag{4.5}$$

subject to the constraint on the batteries level x_b

$$\varphi_1(x,t) = x_{bth} - x_b(t) \leq 0, \tag{4.6}$$

where x_{bth} is a safety threshold.

The letters carried by the robot may also be seen as a constraint and β as a Lagrange multiplier because the functions $\sum_i x_{li}(t)$ and $x_r(t)$ are antagonist: when the former is minimized the latter is maximized. Hence minimizing $\sum_i x_{li}(t)$ and $x_r(t)$ amounts to minimizing $\sum_i x_{li}(t)$ subject to $x_r(t) = 0$. The value of the Lagrange multiplier β is a constant between 0 and 1, so that any contribution to minimize either $\sum_i x_{li}(t)$ or $x_r(t)$ will also minimize $f_1(x,t)$. Moreover it is not necessary to set β to zero when the constraint is satisfied ($x_r(t) = 0$).

The Hamiltonian function

$$F_1(x,\lambda_1,t) = \sum_i x_{li}(t) + \beta x_r(t) + \lambda_1(x,t).\varphi_1(x,t) \tag{4.7}$$

is then deduced and represented graphically (figure 4.6).

A falling edge occurs when the robot

- collects letters from an office;
- posts the letters it holds;
- recharges its batteries and their level is below the threshold (the penalty is removed).

The two concurrent behaviors that are involved in the minimization process of a rectangle's surface are:

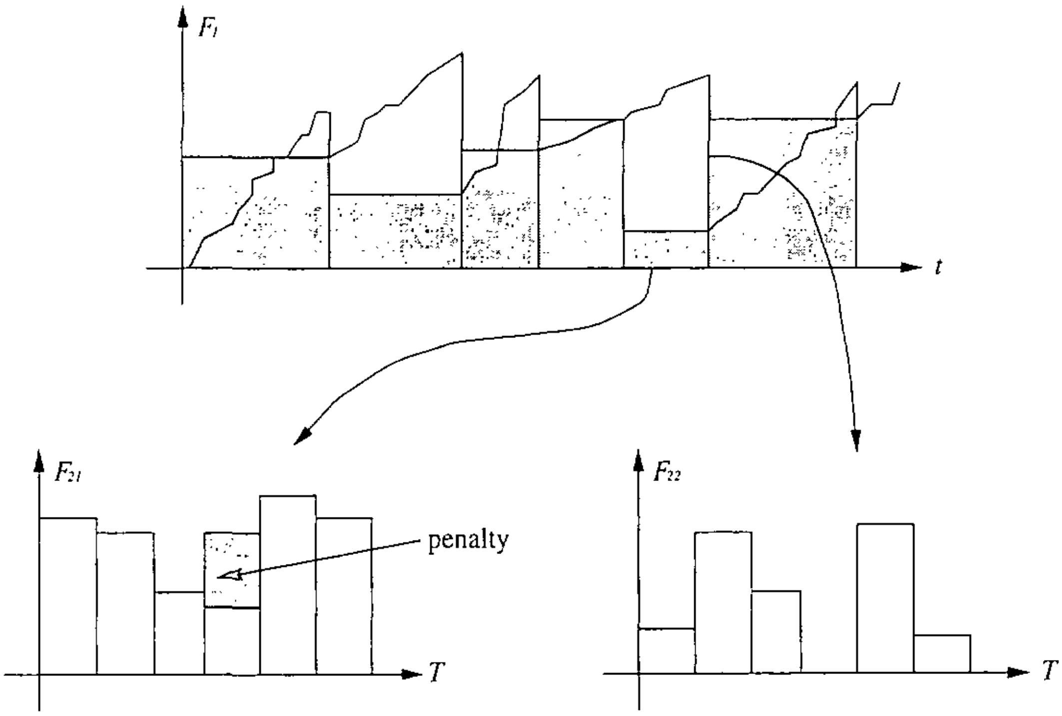

Figure 4.6: The decomposition process of the postman robot problem.

- move to the *nearest* place providing a positive contribution, for the horizontal side;
- move the place providing the *highest* contribution, for the vertical side.

For the first behavior the robot has to minimize the traveled distance x_d between two decision steps $T-1$ and T. The corresponding objective function is

$$f_{21}(x, T) = x_d(T), \tag{4.8}$$

subject to providing a positive contribution (a falling edge in the graphics). In effect, the robot may move to the nearest office but it may not contain any letter. This constraint is expressed by $\varphi_{21}(x, T) = 0$ where

$$\varphi_{21}(x, T) = \begin{cases} 0, \text{ if } F_1(x, \lambda_1, t_T - 1) - F_1(x, \lambda_1, t_T) > 0 \\ 1, \text{otherwise.} \end{cases} \tag{4.9}$$

where t_T is the time step corresponding to the decision step T. We obtain

$$F_{21}(x, \lambda_{21}, T) = x_d(T) + \lambda_{21}(x, T).\varphi_{21}(x, T). \tag{4.10}$$

The second behavior can be defined as the one maximizing

$$F_{22}(x,T) = f_{22}(x,T) = F_1(x,\lambda_1,t_T - 1) - F_1(x,\lambda_1,t_T) \tag{4.11}$$

where t_T is the time step corresponding to the decision step T.

We notice that F_{21} and F_{22} only change at the end of a decision and remain constant the rest of the time. Thus they are represented as a function of T because they depend on the decision step dependent rather than on the time step. Moreover they concern the sides of a single rectangle only. It is the role of the upper behavior to coordinate them in order to minimize the sum of rectangles' surface. This is why a graphical representation of those behaviors does not provide additional information. However it is obvious that the behaviors *nearest* and *highest* may correspond to one of the following five behaviors

- move to an office (3 behaviors);
- move to the mailbox;
- move to the batteries charger;

or more generally to a behavior consisting in moving to a specific place.

Recall from section 3.3.1 that the robot's atomic commands consist of a steering of Δ_θ degrees followed by a translation of Δ_d centimeters. To reach a given goal the robot has to minimize its orientation relative to this goal while moving. This mean that the objective function

$$f_3(t) = x_\theta(t), \tag{4.12}$$

where x_θ is the robot orientation with respect to the goal has to be minimized subject to the obstacles avoidance constraint

$$\varphi_{3i}(x) = (d_{si} - x_{si}) < 0, \tag{4.13}$$

where x_{si} is the robot's reading of sensor i which indicates the distance to the nearest obstacle and d_{si} is the nearest safe distance to an obstacle. The performance criterion we obtain is

$$F_3(x, \lambda_3, t) = x_\theta(t) + \sum_i \lambda_{3i}(x, t).\varphi_{3i}(x, t). \tag{4.14}$$

The hierarchy of sub-behaviors obtained at the end of this stage is sketched in figure 4.7.

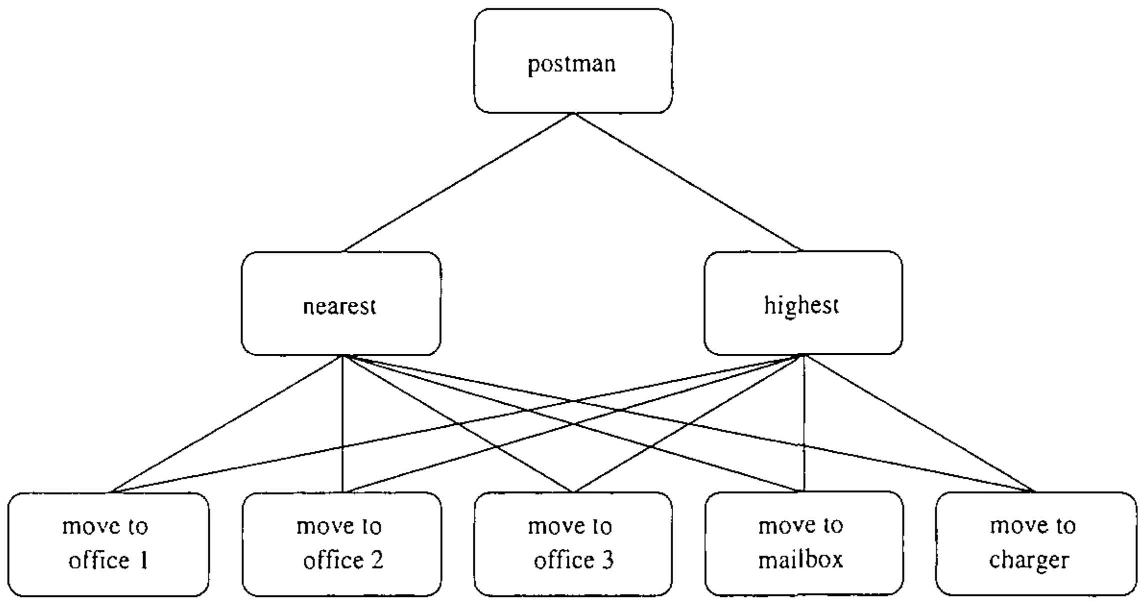

Figure 4.7: The hierarchy of sub-behaviors obtained for the postman robot problem.

4.4.2 Sensory-motor Loop's Design

The design choices that were made for each sensory-motor loop are now described. Each sub-behavior of the hierarchy will be learned using connexionist reinforcement learning. A single MLP with a sigmoid function activation and a single output unit was used to represent the utility function of each command. Some components of the perception vector are represented using a sigmoidal coarse coding as in (Rummery 1995). Basically such a coding works as follows. A number of sigmoid functions, one for each input neuron, are spread across the input space (figure 4.8). As the sigmoid functions overlap each other, each input value will be coded by several values in [0,1] corresponding to the value of each sigmoid function for that input. The input patterns for each network as well as the reinforcement functions are detailed in the experiments section.

Concerning the exploration policy, a simple ϵ-greedy policy was used. A command is chosen according to the probability $P(a = \arg\max_{a \in A(x)} Q(x, a)|x) = 1 - \epsilon$, where ϵ is decreased from 1 to 0 in N_{exp} exploration steps.

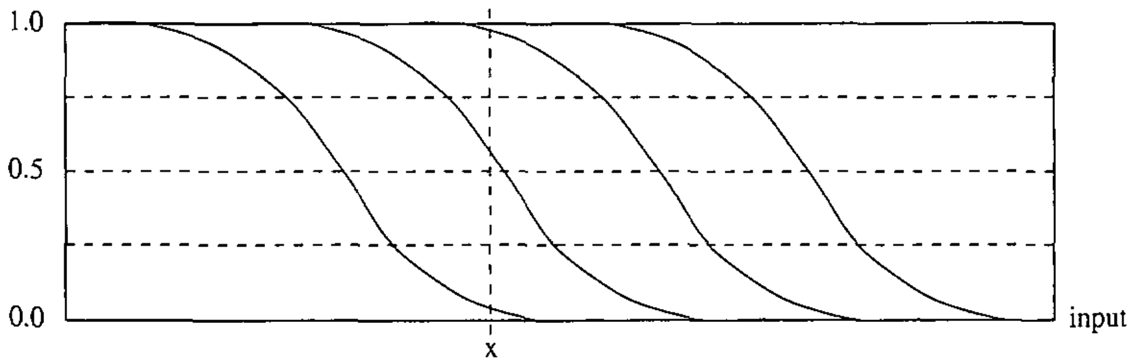

Figure 4.8: The input real-value x is coarse coded into four values in [0,1] which are 0.05, 0.55, 0.95, 1.0 and constitute a suitable input for a neural network.

4.4.3 Coordination

We used a coordination mechanism in which the sensory-motor loops of a given layer are treated as simple commands by the upper level. Once they are activated they keep the control of the agent until they are complete. The control is then returned back to the sensory-motor loop which activated them. This kind of coordination is called Hierarchical Q-learning (Lin 1993).

4.4.4 Evaluation and Validation

To judge the effectiveness of the overall behavior we defined the following metrics:

- the average letters in standby in the offices, the average letters carried by the robot as well as the average batteries level for the external assessment. These values are updated at each interaction cycle (the lowest temporal resolution) to guarantee a uniformity in the comparison with other architectures;
- the average of the global quality criterion, updated at each decision step, to evaluate the learning process.

4.5 Experiments

The postman robot behavior is learned incrementally. With this technique the robot is first trained to learn elementary behaviors and then to learn the upper behaviors using previously acquired skills. This process called modular learning is repeated for each level of the hierarchy. The navigation behaviors are learned separately and preserved using persistent neural networks. The coordination behaviors are then learned so as to achieve the global behavior.

4.5.1 Learning to Navigate

Mobile robot navigation towards a goal while avoiding obstacles has been studied in the reinforcement learning context by Rummery (1995) and Millàn (1996). Their work is an extension of those of Prescott and Mayhew (1992) and Kröse and Van Dam (1993) in which the robot avoids obstacles, not in order to get to a target location, but just to explore the environment. It may be seen as an adaptive construction of a potential field (i.e. the goal generates a potential which pulls the robot towards it, and the obstacles produce a potential which repels the robot away) where the potential vector in a given position is defined by the robot's action with the highest utility in this situation. In classical path planning (Khatib 1986; Barraquand and Latombe 1991) the potential field is computed using *a priori* knowledge about the environment's configuration.

In our experiments recurrent neural networks, with 2 hidden units were used to learn the navigation behaviors. A network's input pattern is a vector of 26 components which are real numbers in the interval [0,1]. The first 16 components correspond to the inverse exponential of distance sensors readings, $e^{-d/k}$ where k is a weighting factor set to 50 during the experiments, and d is a combination of infrared and sonar readings so as to provide measures between 0 and 650 centimeters. The next 8 components are a sigmoid coarse coding of the robot's orientation relative to the goal. The orientation is computed using odometry. The remaining 2 components represent the input context and are linked to the output of hidden units. The input context as well as the orientation allow the robot to differentiate several situations corresponding to the same sensors configuration.

Three commands which are:

- *turn-left* ($\Delta\theta = 22.5^o, \Delta t = 25cm$)
- *turn-right* ($\Delta\theta = -22.5^o, \Delta t = 25cm$)
- *move-forward* ($\Delta\theta = 0^o, \Delta t = 25cm$)

were available to the robot.

The reinforcement function is calculated from the quality criterion $F_3(x, \lambda_3, t) = x_\theta(t) + \sum_i \lambda_{3i}(d_{si} - x_{si})$, without considering the interval between two decisions because the actions have the same duration. We have

$$r_3(t) = F_3(x, \lambda_3, t-1) - F_3(x, \lambda_3, t).$$

Safety thresholds concern only the nine frontal sensors and define a security zone in front of the robot (figure 4.9). We notice that safety thresholds are higher in front of the robot than on its sides. It is simply because the robot can still move even if its sides are near an obstacle but cannot do so if its front is concerned. The values of the Lagrange multipliers when the constraints are violated are chosen to give a penalty which is proportional to the violated surface in the security zone, the overall zone being equivalent to the maximum robot's heading deviation from the goal, which is 180 degrees.

The networks' weights were initialized with random values between -0.1 and 0.1, the discount factor γ was fixed to 0.99, the learning rate α to 2.0, the eligibility trace factor λ to 0.5 and the exploration parameter N_{exp} to 1500 steps. As the networks' output is in the range [0,1], we scaled the reinforcement signal between -0.1 and 0.1 to prevent units from overshooting.

The robot was trained to learn each of the five navigation behaviors in a series of trials, with each trial starting with the robot placed in a different room and ending when it reaches the target location. Figure 4.10 shows the robot's trajectories when it navigates from one room to another, once it has learned. To evaluate the robot learning performances we considered the behavior *move to the charger*. The robot was trained to reach the charger starting from office 3 only. After learning it was able to find the optimal path leading to

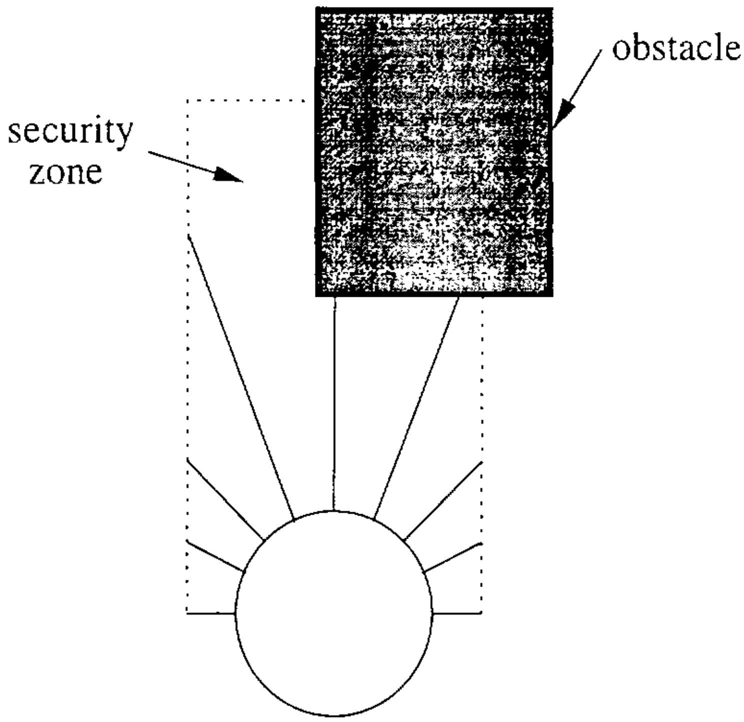

Figure 4.9: The security zone defined in front of the robot.

the charger starting from office 3 (figure 4.11) and also starting from other rooms (figure 4.12), thus exhibiting generalization abilities. Moreover it reacts efficiently to unexpected obstacles (figure 4.13). The learning curves of figure 4.14 show that the robot learns how to move to the charger after 4 trials, corresponding to 6258 steps. However the path found is not optimal and sometimes not safe either. The reason is that during this trial there is a residual exploration of 38%. Thereafter, from the 22^{nd} trial, the path is optimal (between 41 and 44 steps) and safe as we see in figure 4.15 that there are no more penalties after the 22^{th} trial. It is worth adding that during this trial the residual exploration was 16%.

4.5.2 Learning the Coordination

In this section, we report the experiments we carried out to coordinate the navigation behaviors. A grid simulator configured with the distances shown in table 4.2 was used for this purpose.

As shown in the hierarchy of figure 4.7 two intermediate behaviors, *nearest* and *highest*, as well as the global behavior *postman* have to be learned. Once again the robot was first

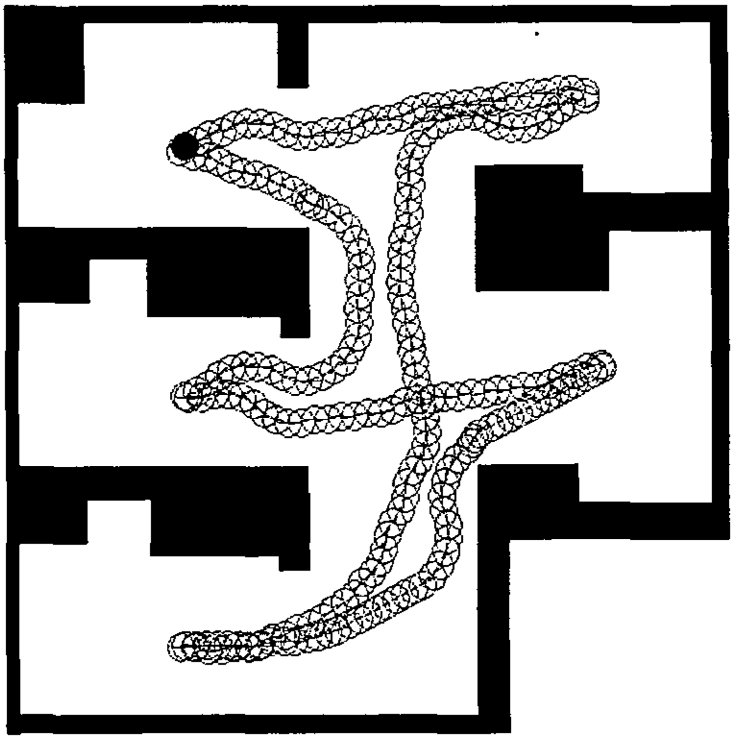

Figure 4.10: The robot moving from one room to another.

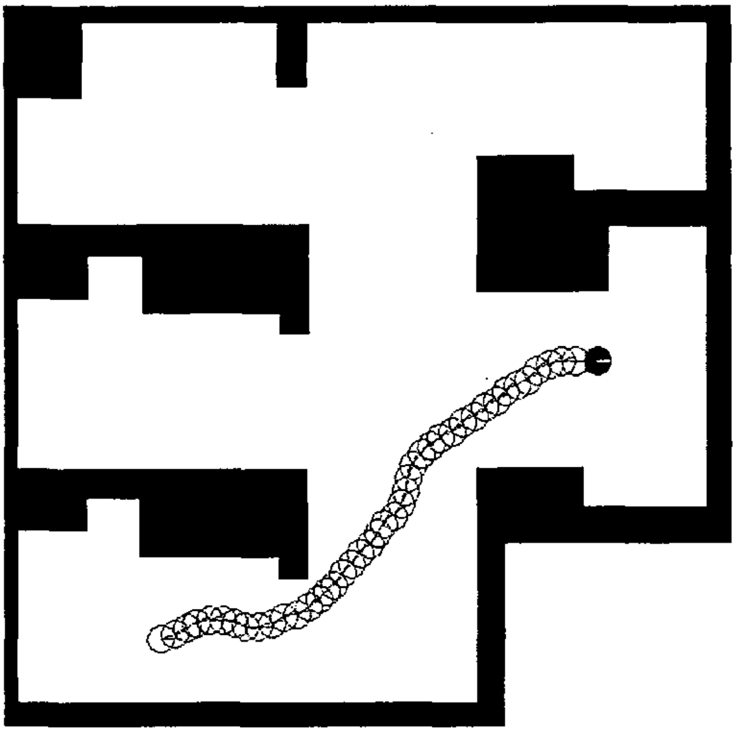

Figure 4.11: The optimal path found between office3 and the charger.

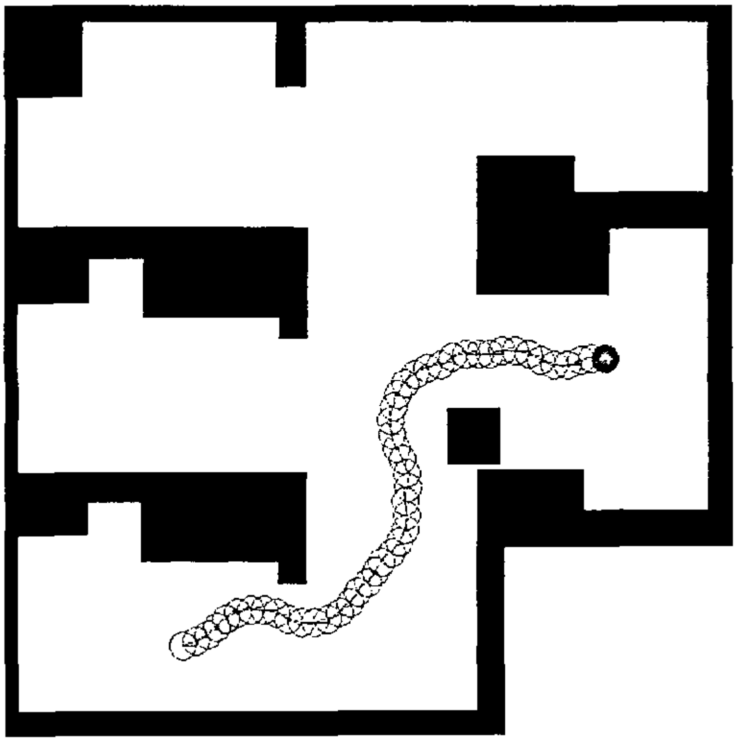

Figure 4.12: Generalization abilities.

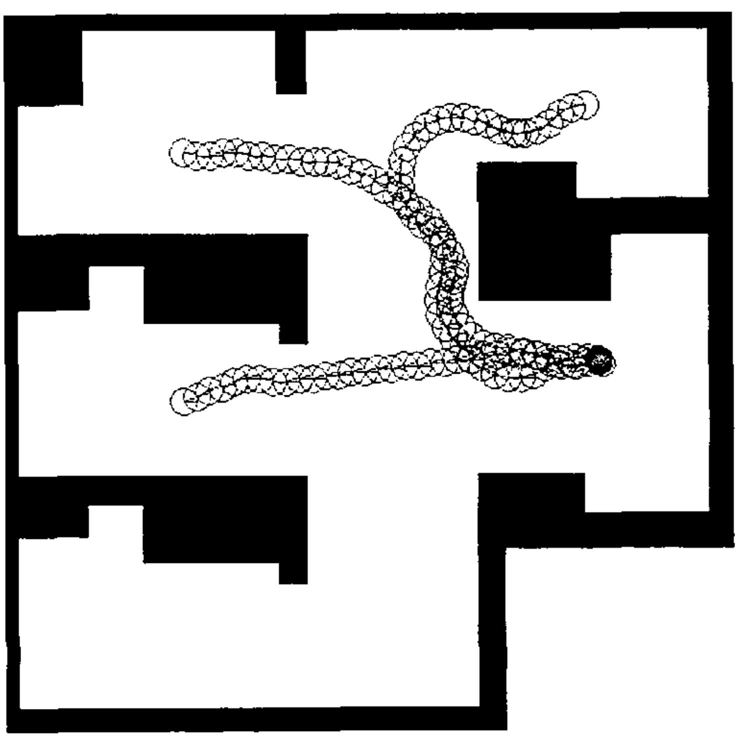

Figure 4.13: Reaction to an unexpected obstacle.

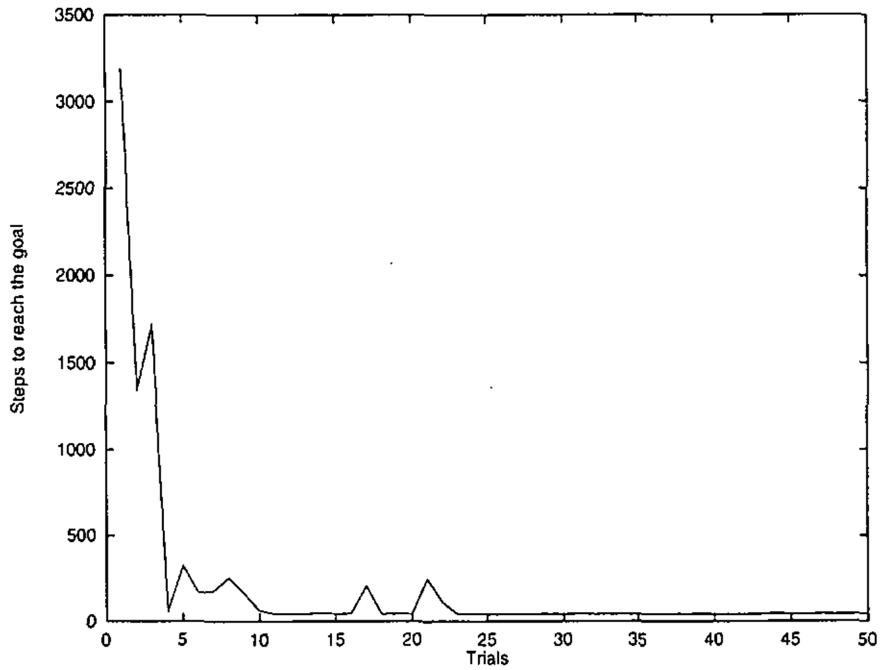

Figure 4.14: Number of steps needed to reach the charger starting from office3 for each trial.

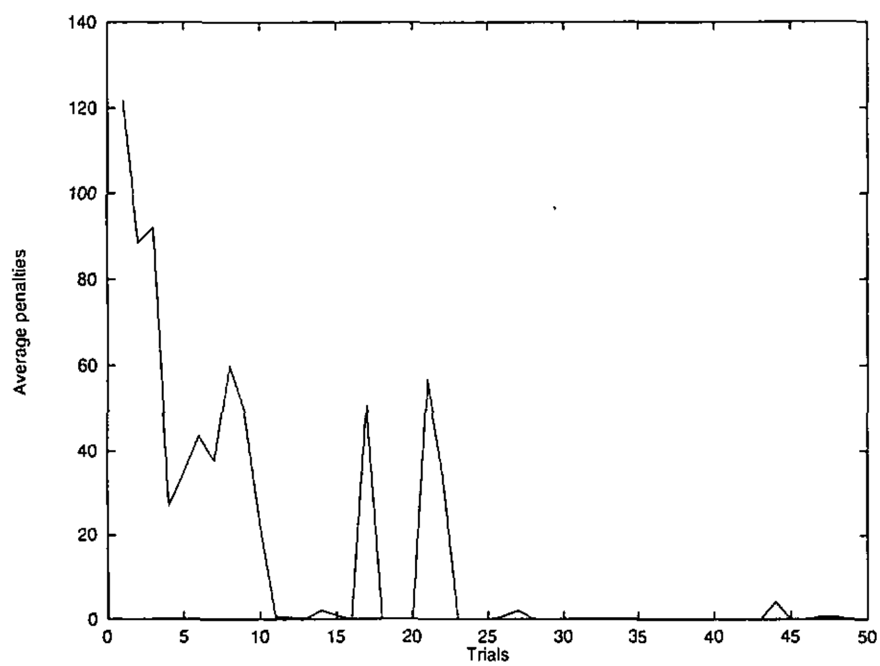

Figure 4.15: Average penalties received during each trial.

	Office 1	Office 2	Office 3	Mailbox	Charger
Office 1	—				
Office 2	39	—			
Office 3	62	41	—		
Mailbox	34	43	65	—	
Charger	40	29	42	44	—

Table 4.2: Steps needed by the robot to move between different places in the environment.

trained to learn the two intermediate behaviors, which were preserved thereafter, and trained to learn the global behavior afterwards. We used feed-forward neural networks to store the Q-values. The same network architecture was used for the three above behaviors, as they share the same state space. It is composed of 40 input units, 3 hidden units and one output unit. All units have a sigmoid activation function. The input pattern is as follows:

- 35 units: each set of 7 units represents a sigmoidal coarse coding of either the number of letters in each office or the number of letters carried by the robot or the batteries level;
- 5 units: each of these units represents a possible location of the robot, i.e. in which place it is. So exactly one unit is 'on' at any decision step.

However the architectures differ in their number of networks and in their reinforcement functions. The intermediate behaviors needed five networks each: one for each navigation behavior. The global behavior needed only two networks: one for each intermediate behavior. The reinforcement function of each behavior is directly computed from the corresponding performance criterion, as explained in section 4.3.3.

The networks' weights, in each sensory-motor loop, were initialized with a random value in the range [-0.1,0.1], and the rest of the parameters as follows: $\gamma = 0.99$, $\lambda = 0.5$, $\alpha = 2.0$ and $N_{exp} = 1500$. Like in the navigation behaviors the reinforcement function was scaled between -0.1 and 0.1.

Since we did not have an optimal policy for the postman robot problem we decided to compare the performance of our hierarchical architecture with those of a flat architecture (figure 4.16) and of a hand-coded controller. In the flat architecture, the high level behavior (*postman*) directly controls the navigation behaviors. It uses the same reinforcement function as the behavior *highest* does. We tried to use a rule based reinforcement function but the results were so bad that it would not be fair to compare them with the hierarchical architecture. So the comparison will be especially made on the architecture rather than on the behaviors' specification, because the behaviors were specified in the same way. The hand-coded controller uses a simple heuristic to choose between the navigation behaviors. This heuristic consists in moving to the office with the highest amount of letters, posting the letters when the amount of letters carried by the robot is higher than the number of letters in each office, and recharging the batteries when their level is below the threshold.

Each of these controllers was tested on 50000 decision steps, a decision step corresponding to an elementary behavior selection, and for each letters flow configuration of table 3.1. The batteries level threshold was set to 60 %.

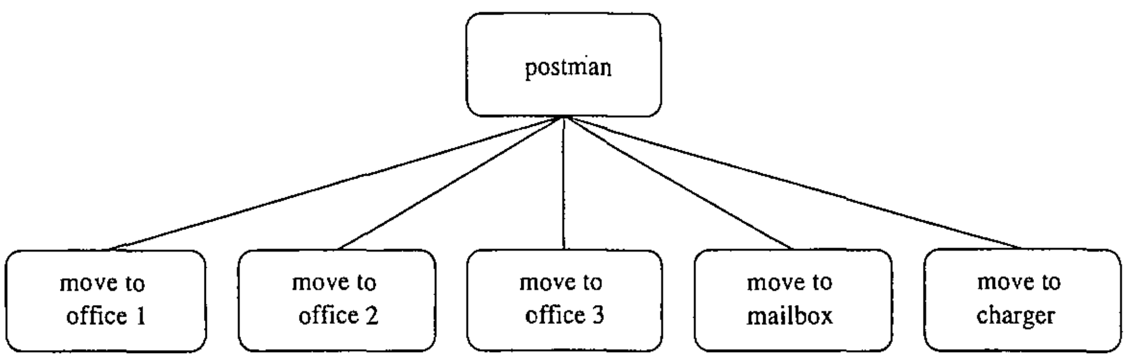

Figure 4.16: The flat architecture used for the comparison with the hierarchical one.

Tables of figure 4.17 show the obtained results. Recall that a good postman robot is the one which minimizes both the number of letters in standby in the offices and the carried letters, while keeping its batteries level above the fixed threshold. We can see that both RL systems achieved good performances compared to those of the hand-coded one. The main reason is that the learning agents implicitly take into account some parameters like the distance between the rooms and the letters flows. Thus they can anticipate the effect of their decisions and move, for example, to the office from which the highest amount of letters will actually be collected. The hand-coded agent decides to move to an office which contains the highest amount of letters, at the moment where the decision is taken but not necessarily when it is completed. On the other hand we notice that the hierarchical architecture outperforms the flat architecture. With the former architecture, there are in average 11.38 and 10.32 (respectively with periodic and Poisson flow) letters in standby in the offices less than with the latter architecture, whereas the average letters carried rises by only 7.56 and 4.23 letters. Moreover a better energy management is achieved by the hierarchical architecture. As it can be observed in the curves of figure 4.18, the hierarchical architecture learns a better strategy than the flat one, and does so very quickly, i.e. it does not behave badly in the beginning. To explain this superiority we argue that the hierarchical architecture explores a smaller search space in the sense that it coordinates only two sensory-motor loops which are pre-learned, whereas the flat architecture coordinates five sensory-motor loops. Another reason is that we were actually solving a Semi-Markov Decision Problem, that is an MDP where the duration of the actions is not the same. The hierarchical architecture takes this feature into account and explicitly considers the elapsed time between two decisions, whereas the flat architecture does not.

4.6 Summary

We have presented a methodology whose objective is to provide helpful guidelines to analyze and design agents capable of solving complex reinforcement learning problems. The methodology must be seen as a conceptual framework in which a number of methods are to be defined. The postman robot case study illustrated how the HPS methodology can be applied. The proposed specification and decomposition methods were successfully tested and have given good results. The methodology must now be applied to solve other prob-

Periodic flow			
Parameters	Hand-coded architecture	Flat architecture	Hierarchical architecture
Average letters in Office 1	11.72	8.49	6.40
Average letters in Office 2	9.87	9.70	4.76
Average letters in Office 3	15.32	17.17	13.95
Average letters carried	16.59	16.42	23.96
Average battery level	68.95	68.92	72.62
Average quality criterion	-46.61	-42.88	-38.18

Poisson distribution flow			
Parameters	Hand-coded architecture	Flat architecture	Hierarchical architecture
Average letters in Office 1	15.52	9.43	8.18
Average letters in Office 2	16.44	13.54	9.72
Average letters in Office 3	20.00	21.97	16.32
Average letters carried	21.09	25.59	29.82
Average battery level	69.78	70.56	77.51
Average quality criterion	-62.10	-55.86	-49.80

Figure 4.17: Tables resuming the performance of the coordination methods for different letters flow configurations.

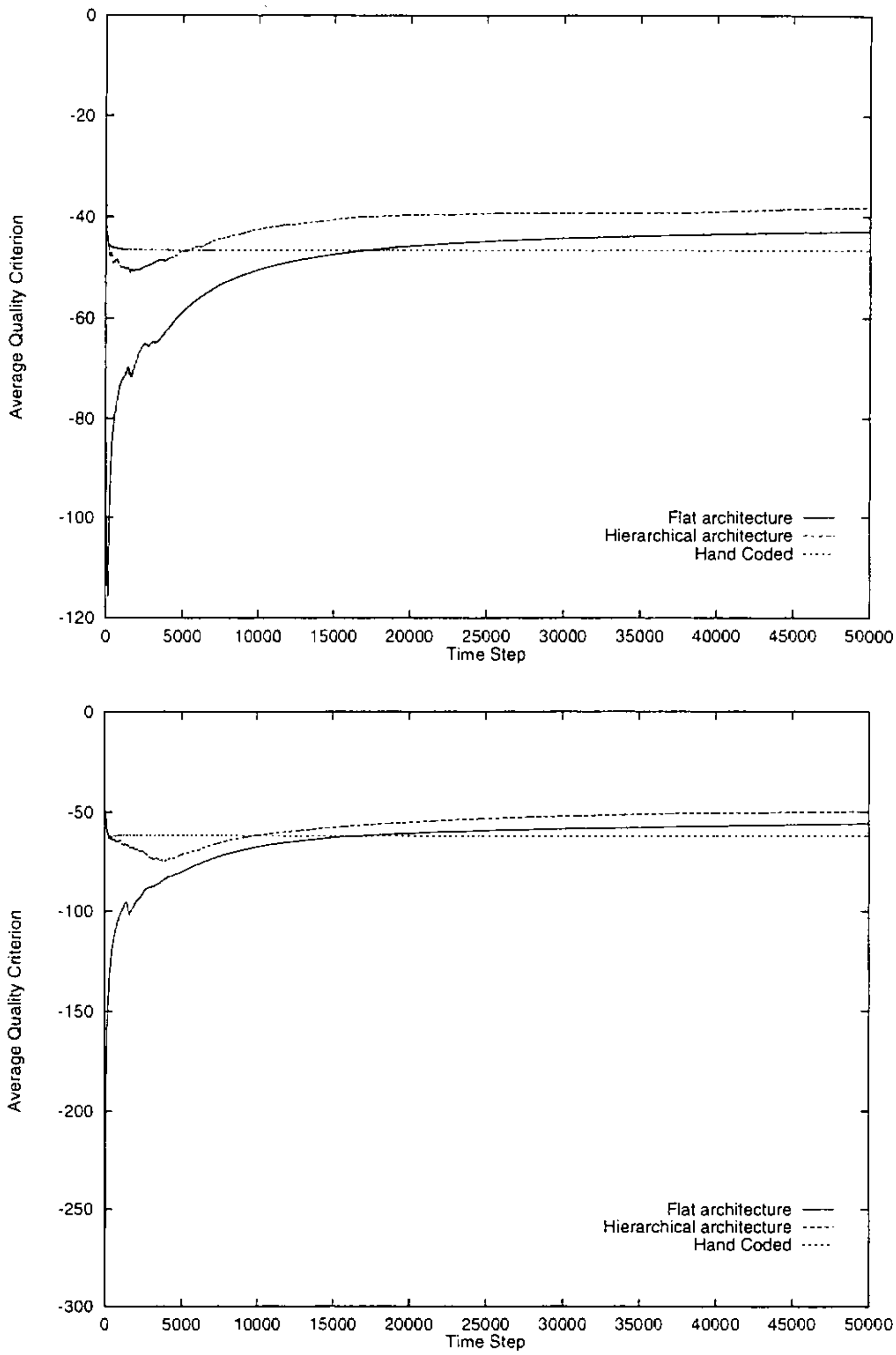

Figure 4.18: Average of the quality criterion as a function of decision steps. The top graph concerns the periodic letters flow and the bottom graph the Poisson distribution letters flow.

lems in order to be generalized and completed, and some efforts need to be done to improve our methods or to propose new ones.

Chapter 5

The Coordination Problem

This chapter concerns the coordination problem, that is, how a complex behavior can be generated by the coordination of several sensory-motor loops. We first review the hierarchical methods that have been proposed so far to scale up reinforcement learning. Then we discuss the properties that a coordination mechanism should have and propose a new coordination method based on the restless bandits theory. Restless bandits allocation indexes are an extension of the Gittins indexes and are borrowed from the field of optimal scheduling. They concern problems involving the sharing of limited resources between several projects which are being pursued. The performances of the proposed method are illustrated through the postman robot problem and compared to those of the Hierarchical Q-learning (Lin 1993).

5.1 Statement

Consider a collection of sensory-motor loops organized in a hierarchical structure (figure 5.1) in which the sensory-motor loops at a given level have a *direct causal influence*, in terms of activation or inhibition, on the sensory-motor loops of the level below. In such a hierarchy, decision making and learning occur at different levels but the interaction with the environment can only take place at the lowest level. Finally, each sensory-motor loop has its own internal state depending on the level at which it intervenes as well as on the task it has to solve. Generally stated, the coordination problem within a hierarchy of sensory-motor loops consists in activating at each time step one sensory-motor loop at

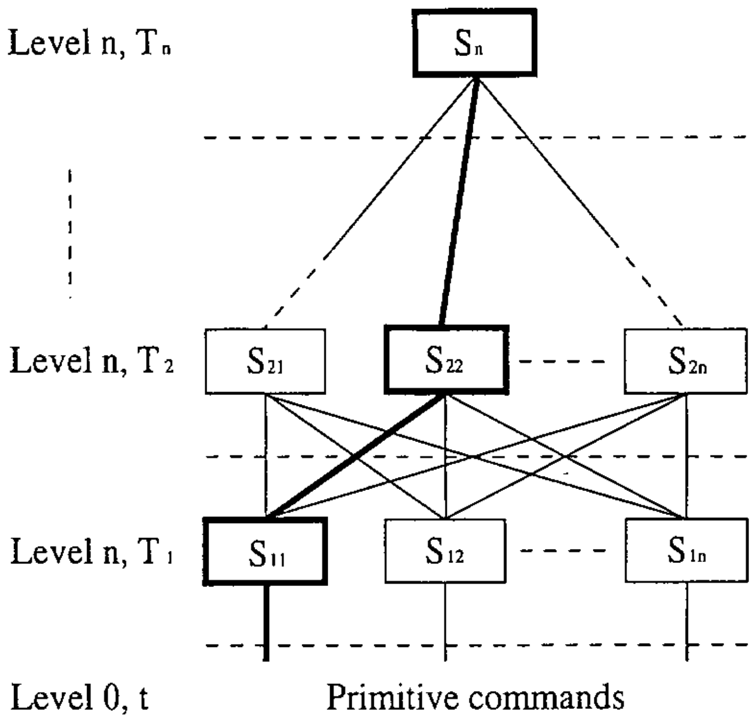

Figure 5.1: A hierarchy of sensory-motor loops. The path of active sensory-motor loops on a given time step is represented in bold.

each level in order to generate the global expected behavior. This is also known as the *action selection problem* and concerns the resolution of conflicts which arise when several actions or behaviors compete to access to limited motor resources. It has been studied in ethology (McFarland 1981) as well as in adaptive behavior (Tyrell 1993).

5.2 Related Work

It has been recognized that the use of hierarchies in reinforcement learning improves the learning performances. It allows a better exploration of the search space, the reuse of previously learned skills at a given level to acquire new skills at the level above, and speed up the overall learning process. Although we are especially interested in the selection device, that is, the mechanism that allows switching between sensory-motor loops, we take the opportunity to review most of the work done in hierarchical reinforcement learning. This work can be roughly grouped into four categories:

1. command and temporal abstraction;

2. state abstraction (coarsening or aggregation);

3. MDP decomposition (state space partitioning);

4. sub-goals decomposition (modular approaches).

Of course there may be approaches that fall in multiples categories.

5.2.1 Hierarchical Q-Learning

When we think about coordination methods the first idea that comes to mind is to treat the sensory-motor loops as primitive commands. This approach has been first introduced by Mahadevan and Connell (1992). In their work a global behavior consisting in box-pushing was decomposed into elementary sub-behaviors (finder, pusher, unwedger) which were learned independently using reinforcement learning. A hand-coded arbiter switches between the sub-behaviors according to their applicability conditions and their precedence.

Lin (1993) went further and proposed a system in which both the sub-behaviors and the arbiter were learned using Q-learning. The task to be achieved consisted of finding a batteries' charger in an office environment and connecting to it. As this task is so difficult to learn by a monolithic agent, it was decomposed into three sub-behaviors: following walls on the robot's left/right hand side, passing a door, docking on the charger. Each sub-behavior Sb_i was learned by a single sensory-motor loop with Q-learning using a local reinforcement function. These new skills are then used as actions by the arbiter which learns $Q(state, Sb_i)$ with global reinforcement function and state space. A sub-behavior is selected according to its Q-value and some applicability conditions, and a new decision is made when an active sub-behavior ends or another one becomes applicable.

5.2.2 Feudal Q-Learning

The principle of this approach, proposed by Dayan and Hinton (1993), is to operate a coarsening at each level of the hierarchy, that is, each state at a given level represents an aggregation of states at the immediately lower level. The goal state is also abstracted so that for each level i, the goal state is the one to which the goal state at the lower level

$i-1$ belongs. Given that each level has an assigned manager, the learning procedure works as follows. The manager of a level i being in abstract state S_1^i performs a command C^i which should lead him to state S_2^i. This command becomes a goal for the manager of the lower level $i-1$, in the sense that commands have to be executed in order to enter a state S_2^{i-1} in the aggregation represented by S_2^i. This procedure continues until the lowest level where a primitive command is executed. An abstract action ends when a new state is observed at the same level. A manager is then rewarded if he achieves his goal and punished otherwise. If the goal is reached at a given level, its manager delegates the responsibility to his sub-manager to search within the zone defined by his abstract state. This approach was applied to a robot navigation task in 8×8 grid without obstacles.

It has recently been extended by Dietterich (1997) who adds the possibility of hierarchical learning of the Q-values. The value function of an abstract command (i.e., the sum of rewards generated by the execution of this abstract commands) is treated as an immediate reward by the level that selects it, just as the first level does with primitive commands. The direct consequence of this improvement is the *polling execution* of the hierarchy, that is, a decision is made at each level at each time step.

5.2.3 Hierarchical Distance to Goal

MDP decomposition methods consist in partitioning the state space into regions and computing an optimal policy for each of them. The resulting policies are then combined to solve the initial MDP.

In the HDG algorithm (Hierarchical Distance to Goal) proposed by Kaelbling (1993a) the state space is partitioned so that each region corresponds to a landmark. A landmark is actually a specific state and a region is composed by states that are closer to a landmark than to any other one. First a high-level policy that leads to the goal region (i.e. the region containing the goal state) starting from any other region is learned. It gives the agent the next region to reach on the route from its current region (i.e. the agent closest's landmark) to the goal region. Then for each region, a policy that allows the agent to move to the neighboring region is learned. Once the agent is in the goal's region, it learns how to reach the goal state. The union of these policies defines the global solution.

The landmarks are given *a priori* by the designer. However methods to autonomously find them are currently being investigated.

Similar approaches have been studied by Parr (1998), Dean and Lin (1995) and Hauskrecht et al. (1998).

5.2.4 W-Learning

In modular approaches, sensory motor loops are used as gating mechanisms whose role is to control the flow of commands from the bottom to the top of the hierarchy. There is no temporal or state abstraction. The problem is solved at the lowest level of abstraction by using suggestions of several experts. Humphrys (1997) and Whitehead et al. (1993) proposed a two-level architecture in which several modules (sensory-motor loops) compete to get the control of the agent. Each module learns how to achieve a sub-goal and maintains its own Q-values tables. In a given state x observed by the agent, each module M_i suggests a command c_i it wants to see executed. The module chooses the command according to its utility $Q_i(x, c_i)$ and strengthens it with a weight $W_i(x)$. The agent finds the module M_k with the highest weight

$$W_k(x) = \max_i W_i(x),$$

and executes the suggested command c_k. The value of $W_i(x)$ may be computed as follows:

- $W_i(x) = Q_i(x, c_i)$ called *maximize best happiness* by Humphrys (1997), and *nearest neighbor* by Whitehead et al. (1993);
- $W_i(x) = \sum_i Q_i(x, c_i)$ called *maximize collective happiness* by Humphrys (1997), and *greatest mass* by Whitehead et al. (1993).

A more interesting way to compute $W_i(x)$ is to make it express the difference between the utility $Q_i(x, c_i)$ that a module M_i has of being obeyed and the utility $Q_i(x, c_k)$ of not being obeyed (actually following the suggestion of module M_k):

$$W_i(x) = Q_i(x, c_i) - Q_i(x, c_k).$$

This approach similar to the ours which we introduce in section 5.4.1. The idea is to *minimize the worst unhappiness*, that is, perform the command c_k of the module M_k that will most suffer if it is not obeyed:

$$W_k(x) = \max_j \max_i (Q_i(x, c_i) - Q_i(x, c_j)).$$

However, the result of tbe selection is greatly influenced by the order in which the modules' suggestions are examined, and the same suite of commands is needed for all modules.

To overcome this drawback, Humphrys (1997) proposed the following update rule, which he called *W-learning* to estimate $W_i(x)$ online, even when the modules do not share the same set of commands:

$$W_i(x) \leftarrow (1 - \alpha)W_i(x) + \alpha(Q_i(x, c_i) - (r_i + \gamma \max_{b \in C_i} Q_i(y, b)),$$

for all $i \neq k$ where M_k is the winning module. We notice that the transition is caused by the command c_k and that the error represents the loss of profit of module M_i. It is assumed in this rule that Q_i is already learned. Therefore if Q_i and $W_i(x)$ are to be estimated conjointly, then it is necessary to delay the learning of $W_i(x)$.

5.2.5 Compositional Q-Learning

Singh (1992) developed an architecture to solve compositional tasks, that is, tasks which can be expressed as a sequence of sub-tasks. The originality of his approach is that sub-tasks are not *a priori* assigned to sensory-motor loops. During the learning phase a reward is generated only when a sub-task is achieved or when the whole composite task is completed. A gating function learns to select the sensory-motor loop that will actually perform its command. The winning sensory-motor loop is the one who has the best estimate of the Q-values (or has the smallest expected error) of the sub-task that is currently executed. Because the sensory-motor loop that has produced the least error learns the most (in proportion to the error), the more a sensory-motor loop learns a given sub-task, the more it improves its Q-values estimate. Thus its probability of being selected for the same sub-task will increase leading to the emergence of sub-task assignment over sensory-motor loops.

5.2.6 Macro Q-Learning

Finally, Sutton et al. (1998) studied the case where an MDP has to be solved using abstract actions (*options* or *macro-actions* as they called them). To do so they used *SMDP Q-learning* (Bradtke and Duff 1995; Mahadevan et al. 1997) and introduced the notion of *Termination Improvement.* During the execution of a particular option o, launched at time t from state s_t and normally terminating at time $t+k$, it is possible to update the utility values of performing option o (as well as other options whose trajectories are included in the one of option o) from each state s_{t+i} $(1 \leq i < k)$. Thus, information to make decisions is available in every state and an ongoing option can be interrupted in any state in favor of a more promising option. The notion of macro-action interruption is discussed in the next section.

5.3 The Selection Device

To ensure efficient coordination of sensory-motor loops a number of useful characteristics are required for the selection device (as reported by Prescott et al. (1999)).

Providing *clean switching*, that is directly selecting the sensory-motor loop with the highest activation, constitutes the first property. The second one states that there must be no interference from sensory-motor loops that are applicable but not selected, in other words, only the selected sensory-motor loop controls the agent. These two properties can be implemented by an *indexed policy* (see the next section). To define the third characteristic we first need to introduce the notion of *preemption.* The approaches reviewed in the previous section can be classified into two categories: those using a *command selection* scheme and those using a *behavior selection* scheme. These two schemes are respectively preemptive and non-preemptive.

In the behavior selection scheme, the learning process is uniform through the levels of the hierarchy and the problem is solved at different levels of abstraction. However from the second level of the hierarchy on, the action selection is replaced by the behavior selection and the time scale for decision making rises from t to T_1 (figure 5.1). This means that once a sensory-motor loop is selected it will keep the control of the agent until its completion[1]. Hence learning will occur only at the termination of this sensory-motor loop. Because a selected sensory-motor loop cannot be interrupted, this scheme may have some drawbacks in problems involving the satisfaction of multiple and concurrent objectives. On the other hand exploration is improved because the state space is covered using big steps (Dietterich 1997).

In the action selection scheme only one behavior will remain: it is the one which is produced by the overall system. It may be analyzed at various levels as consisting of streams of actions ranging from reactive to planning operations. At each time step the system learns and makes decisions at each level of the hierarchy. Therefore any sensory-motor loops at any levels may interrupt each other. Such a continual interruption leads to inefficient exploration because it reduces the probability of reaching sensible states.

Let's illustrate these two schemes by the traditional ethological example of an animal having to satisfy both hunger and thirst drives. We assume that food and water are in different locations and that there are several levels of thirst and hunger. Suppose that the animal is hungry and that this activates the behavior leading it towards the food. If the thirst level becomes higher than the hunger one and the animal cannot interrupt the selected behavior it might die of dehydration en-route towards the food. On the other hand if it can interrupt its behaviors at any time and the levels of thirst and hunger become alternatively higher and lower one relative to the other, it may die of starvation or dehydration somewhere between the two locations.

These two approaches seem to be extremist but complementary. To find a compromise we can either introduce a model of fatigue, which is based on time sharing, to the action selection scheme or allow the interruption of sensory-motor loops in the behavior selection scheme. The second method seems to be more natural than the first one but may exhibit an unstable behavior. In effect, two sensory-motor loops with closed activation degree may interrupt each other (as explained in the above example), thus generating an oscillation. This phenomenon is called *dithering* (Prescott et al. 1999; Redgrave et al. 1999). A way of

[1] A sensory-motor loop is completed when it reaches a state which is a goal or in which it is not applicable anymore.

overcoming this problem is to add some kind of *persistence* to the active sensory-motor loop. It means that to interrupt an active sensory-motor loop, the candidate sensory-motor loop (i.e. the one with the highest activation degree among the applicable but inactive sensory-motor loops) must not only have a greater activation degree than the active sensory-motor loop has but must also exceed it by a given constant $w/2$. The constant w is the width of the hysteresis loop representing the behavior switching between active and passive phases (figure 5.2).

It has been hypothesized that the selection mechanism of this form is implemented in the vertebrate brain by the *Basal Ganglia* (Prescott et al. 1999; Redgrave et al. 1999).

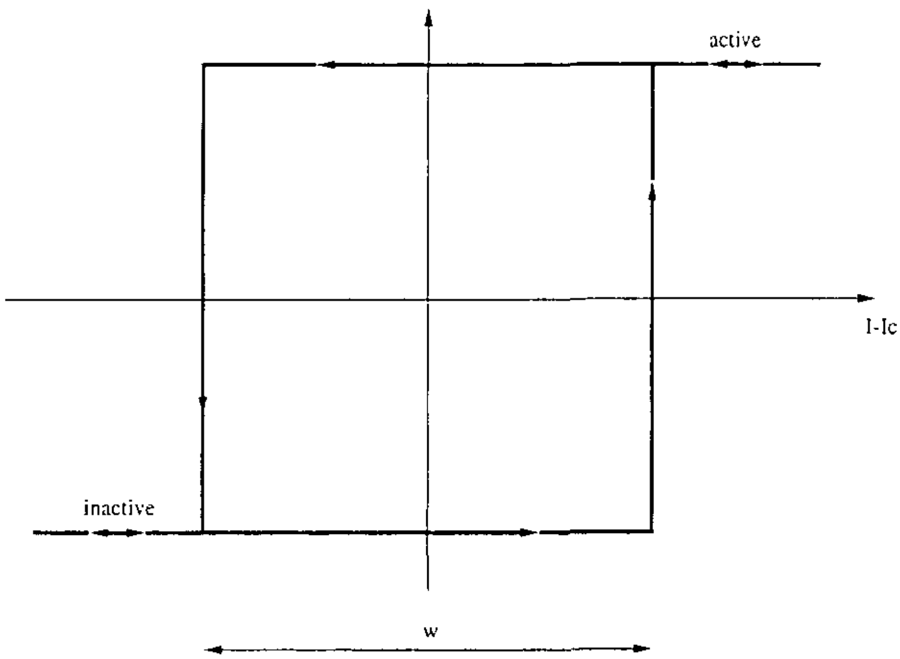

Figure 5.2: The hysteresis loop representing the behavior switching between the active and passive phases. Ic is the index of the candidate behavior and w is the width of the hysteresis.

5.4 Indexed Policy

An indexed policy consists in allocating an index to each sensory-motor loop and to activate the one with the highest index in a winner-take-all manner. Of course indexes which are computed adaptively and on-line are highly desirable. In hierarchical Q-learning (Lin 1993) the indexes simply correspond to the Q-values of selecting a sensory-motor loop in a certain global state. In W-learning the index is the value of the strength $W(x)$. Here we introduce another method to compute such indexes, based on the restless bandit theory, which we call *RBI-learning*.

5.4.1 The Restless Bandits

The restless bandits are an extension of the multi-armed bandit problem and have been studied by Whittle (1988), and Weber and Weiss (1989). The initial problem concerns n projects, the state of a project i at time t being denoted by $x_i(t)$. At each time step t only one project has to be operated. If the operated project is i then it will generate a reward $r_i(t)$ and make a transition $x_i(t) \to x_i(t+1)$ according to its transition probabilities P_i. The other $n-1$ projects remain frozen, i.e. neither produce reward nor change state. A project is said to be in an *active* or *passive* phase depending upon whether it is selected or not. Gittins (1989) has shown that an index policy is optimal for this problem. Such an index is denoted $I_i(x_i)$ and is a function of the project i as well as its state x_i:

$$I_i(x_i) = \max_{\tau>0} \frac{E\left[\sum_{t=0}^{\tau-1} \gamma^t r_i(t)\right]}{E\left[\sum_{t=0}^{\tau-1} \gamma^t\right]}. \tag{5.1}$$

This index can be interpreted as the maximal value of the reward density relative to the stopping time τ. The optimal policy will simply be to select the project with the greatest index. The nice property of such a strategy is that I_i only depends on information concerning project i. The dimensionality of the problem is considerably reduced.

To have a better and intuitive understanding of the Gittins' indexes, we will examine the following didactic example provided by Duff[2], where for the sake of simplicity the rewards are deterministic. Imagine several stacks containing numbers, which are rewards, and suppose that we can see the entire contents of each stack. Our goal is to pop the stacks in an order that maximizes the discounted sum of the resulting reward stream. We can convince ourselves that the optimal strategy involves popping the stack with the highest reward density:

$$D_i = \max_{T} \frac{\sum_{k=0}^{T-1} \gamma^k r_i(k)}{\sum_{k=0}^{T-1} \gamma^k}, \tag{5.2}$$

[2]Personal communication

where $r_i(k)$ is the contents of stack i in position k, starting from the top. Stacks with higher reward density contain high rewards near their top and have to be popped first because of the discount factor (figure 5.3).

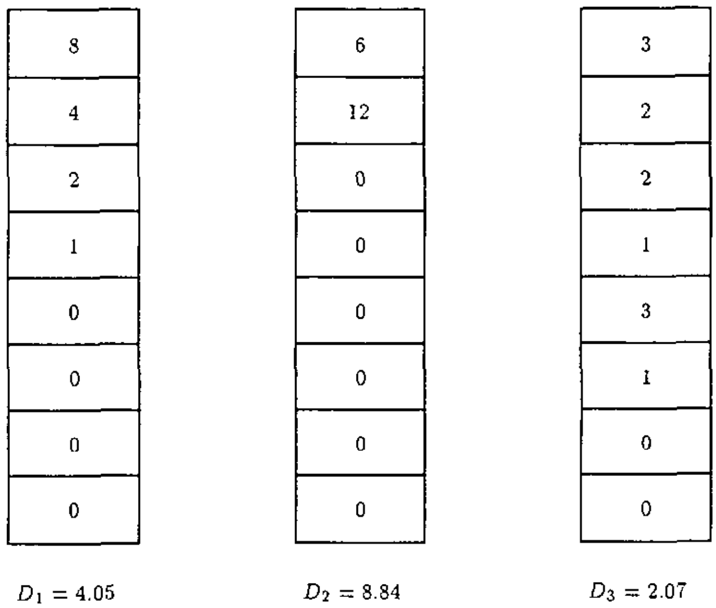

Figure 5.3: Stacks reward densities for $\gamma = 0.9$. Notice that the stack to pop is not necessary the one with the highest value at its top.

Unfortunately this method cannot be directly applied to solve the coordination problem (the projects[3] being replaced by sensory-motor loops) because the fundamental assumption (i.e. the unselected projects remain frozen) is not valid anymore. This happens in many cases and especially in mobile robotics because the states of the sensory-motor loops are built from the same agent's perceptions and these perceptions evolve whatever the selected sensory-motor loop is.

To treat the restless bandits problem we will introduce the following notation:

- X_i is the set of states of project i;
- $P_i(x, y, k)$ is the probability that project i moves from state x to state y when it is in phase k, where $k = 1$ or 2 for respectively the active or the passive phase;
- $r_{ik}(t)$ is the reward produced at time t by project i in phase k.

[3]To introduce the theory we will use the term project instead of sensory-motor loop.

If we want to maximize the discounted sum of reward over an infinite horizon, for a single project i we just have to solve the following optimality equation

$$V_i(x) = \max_{k=1,2} E\left\{ r_{ik} + \gamma \sum_{y \in X_i} P_i(x,y,k) V_i(y) \right\}, \tag{5.3}$$

where $V_i(x)$ is the value function of project i in state x. To do so we will compute the Q-values

$$Q_i(x,k) = E\left[r_{ik} + \gamma \max_{b=1,2} Q_i(y,b) \right] \tag{5.4}$$

and then decide to activate or freeze the project according its Q-values.

Consider now the multi-project case. We are essentially interested in maximizing

$$\sum_t \gamma^t \sum_i r_{ik}(t) \tag{5.5}$$

subject to $\sum_i l_i(t) = n-1$ where $l_i(t) = 1$ if project i is passive at time t and $l_i(t) = 0$ otherwise (it means that at each time step only one project has to be active). Such a maximization amounts to maximizing

$$\sum_t (\gamma^t \sum_i r_{ik}(t) + \nu(t) \sum_i l_i(t)) \tag{5.6}$$

where ν is a Lagrangian multiplier. The new optimality equation to solve becomes

$$V_i(x) = \max_{k=1,2} \left\{ r_{ik} + \nu l_i + \gamma \sum_{y \in X_i} P_i(x,y,k) V_i(y) \right\}, \tag{5.7}$$

or more compactly

$$V_i(x) = \max\left[L_{i1}, \nu + L_{i2} \right], \tag{5.8}$$

where

$$L_{ik} = r_{ik} + \gamma \sum_{y \in X_i} P_i(x,y,k) V_i(y). \tag{5.9}$$

Whittle said that ν can be seen by an economist as a *'subsidy for passivity'* tuned at a level which guarantees that only one project is active at a time. The index of a project i in state x_i is then defined as being the value $\nu_i(x_i)$ of ν which makes $L_{i1} = \nu + L_{i2}$. It can be computed by using the Q-values of a project.

Proposition The index of a project i in state x_i is:

$$\nu_i(x_i) = Q_i(x_i, 1) - Q_i(x_i, 2).$$

Proof Let $x = (x_1, x_2, ..., x_i, ..., x_n)$ be the composite state of the global problem, and let $Q(x, k)$ be the utility of activating project k in state x:

$$Q(x, k) = Q_k(x_k, 1) + \sum_{i \neq k} Q_i(x_i, 2)$$

Let m be the project that maximizes this utility. We have:

$$Q(x, m) = \max_k Q(x, k) \Rightarrow Q(x, m) \geq Q(x, k) \quad \forall\, k \in [1, n]$$

This inequality can be written as follows:

$$\begin{aligned}
& Q_m(x_m, 1) + Q_k(x_k, 2) + \sum_{i \neq k,m} Q_i(x_i, 2) \geq Q_m(x_m, 2) + Q_k(x_k, 1) + \sum_{i \neq k,m} Q_i(x_i, 2) \\
\Rightarrow \quad & Q_m(x_m, 1) + Q_k(x_k, 2) \geq Q_m(x_m, 2) + Q_k(x_k, 1) \\
\Rightarrow \quad & Q_m(x_m, 1) - Q_m(x_m, 2) \geq Q_k(x_k, 1) - Q_k(x_k, 2). \quad \text{Q.E.D.}
\end{aligned}$$

Figure 5.4 shows the algorithm of RBI-learning.

5.4.2 Discussion

Intuitively we can see that the index actually reflects the need for a project to be active with respect to the exploration and exploitation criteria. Actually the value of ν increases if

- $Q_i(x_i, 1)$ increases which means that the project needs to be active (exploitation phase), or
- $Q_i(x_i, 2)$ decreases which means that the project does not want to be passive (exploration of the effects of its activation). This condition holds as far as the project is deteriorating during its passive phase (i.e., receiving negative rewards).

```
loop
  Observe state x_i for each project i

  for each project i do
    I_i(x_i) = Q_i(x_i, 1) - Q_i(x_i, 2)
  end for

  Activate project k such that I_k(x_i) = max_i I_i(x_i)

  Update Q_k(_ix, 1)
  for each project i ≠ k do
    Update Q_i(x_i, 2)
  end for
end loop
```

Figure 5.4: Algorithm of RBI-learning.

On the other hand the utility to a project of being active or passive can be seen respectively as activation and inhibition signals. Thus, persistence may be implemented by simply removing the inhibition signal from the selected sensory-motor loop and keeping it for others.

Our coordination method may be situated between hierarchical Q-learning and W-learning. RBI-learning and W-learning are similar because they are both motivated by the same criterion, which is to reduce the loss of profit when a project (module) is not selected (obeyed). However they differ in the sense that RBI-learning supports temporal abstraction (like hierarchical Q-learning) whereas W-learning does not. Actually W-learning needs to perform an update after each execution of a primitive command. In addition RBI-learning is supported by a strong theory and does not require any pre-learned Q-values.

5.5 Experiments

The coordination method we presented above is now evaluated and its performances compared to those of Hierarchical Q-learning[4]. To do so we have followed the HPS methodol-

[4]We have not made any comparison with W-learning. The reason is that it is not applicable to the postman robot problem because it does not support temporal abstraction. In effect, the update of Q-values

ogy therefore some results and settings from the previous chapter are reused. We decided to test these coordination methods on the flat architecture. using the behavior selection scheme. The network architecture used to implement the Hierarchical Q-learning method is the same as in the previous chapter. However we view it here. Each of the five neural networks of the architecture is composed of 40 input units, 3 hidden units and one output unit. All units have a sigmoid activation function. The input pattern is as follows:

- 35 units: each set of 7 units represents a sigmoidal coarse coding of either the number of letters in each office or the number of letters carried by the robot or the batteries level;
- 5 units: each of these units represents a possible location of the robot, i.e. in which room it is. So exactly one unit is 'on' at each decision step.

Recall also that the function to be optimized is $f(x, \lambda, t) = \sum_i x_{li}(t) + \beta x_r(t) + \lambda_1(x, t)(x_{bth} - x_b(t))$ and the instantaneous reinforcement function is $r(t) = f(x, \lambda, t) - f(x, \lambda, t-1)$. For the RBI-learning. the above function is linearly decomposed into five functions: one for each elementary sensory-motor loop. We obtained:

- $f_1(x, t) = x_{l1}(t)$ for the sensory-motor corresponding the behavior *move to office1*;
- $f_2(x, t) = x_{l2}(t)$ for the sensory-motor corresponding the behavior *move to office2*;
- $f_3(x, t) = x_{l3}(t)$ for the sensory-motor corresponding the behavior *move to office3*;
- $f_4(x, t) = \beta x_r(t)$ for the sensory-motor corresponding the behavior *move to the mailbox*;
- $f_5(x, \lambda, t) = \lambda_1(x, t)(x_{bth} - x_b(t))$ for the sensory-motor corresponding the behavior *move to the charger.*

We used two different network architectures to implement the restless bandits method. In the first one all the sensory-motor loops share the same state space so it is similar to the one

of primitive commands (robot's movements) would be inefficient because the state space is huge and the reinforcement is only given when the robot reaches one of the sub-goals (offices, mailbox, charger).

of Hierarchical Q-learning. In the second architecture, the state space is reduced for each sensory-motor loop in order to keep only features relevant to the function to be optimized. Thus, for each sensory-motor loop, we kept features representing the robot location (5 units) and features representing the amount to be optimized (7 units), corresponding for example to the number of letters carried by the robot for the behavior *move to the mailbox*. Hence we obtained networks with 12 input units, 2 hidden units and one output unit. However, for both architectures we needed two networks for each sensory-motor loop: one to approximate the Q-values for each phase (passive or active) of the sensory-motor loop.

The networks weights, in each sensory-motor loop, were initialized with a random value in the range [-0.1,0.1] and the reinforcement function was scaled between -0.1 and 0.1. The rest of the parameters was set as follows: $\gamma = 0.99$, $\lambda = 0.5$, $\alpha = 2.0$ and, for Hierarchical Q-learning $N_{exp} = 1500$. There was no exploration phase for RBI-learning.

Each of these controllers was tested on 50000 decision steps, a decision step corresponding to an elementary behavior selection, and for each letters flow configuration of table 3.1. The batteries level threshold was set to 60 %.

The results reported in the tables of figure 5.5 show that RBI-learning outperforms Hierarchical Q-learning. For the Poisson distribution we can see that with the former method, there are in average 8.75 letters less in standby in the offices than with the latter method, whereas the average of carried letters increases by only 5.88 letters. For the periodic flow the carried letters are almost the same whereas the letters in standby drop by 5.33 letters for the RBI-learning. Moreover a better energy management is achieved by the restless bandits method. The performances of RBI-learning can be justified by the fact that the reinforcement function is decomposed and that there is a good balancing between exploration and exploitation which allows a good strategy to be found very quickly (figure 5.6).

Surprisingly, the RBI-learning with reduced state space did not give the expected results. We expected that, because of the small search spaces, a better strategy would have been found or at least the indexes would have been learned more quickly. It seems that in practice, better performances are to be expected when the state space is the same[5].

Periodic flow			
Parameters	Hierarchical Q-learning	Restless Bandits full space	Restless Bandits reduced space
Average letters in Office 1	8.49	8.88	9.82
Average letters in Office 2	9.70	8.26	16.65
Average letters in Office 3	17.17	12.89	18.47
Average letters carried	16.42	15.31	16.18
Average battery level	68.92	71.00	71.42
Average quality criterion	-42.88	-37.42	-53.06

Poisson distribution flow			
Parameters	Hierarchical Q-learning	Restless Bandits full space	Restless Bandits reduced space
Average letters in Office 1	9.43	7.75	10.15
Average letters in Office 2	13.54	11.21	17.99
Average letters in Office 3	21.97	17.23	21.00
Average letters carried	25.59	31.47	27.82
Average battery level	70.56	72.12	68.17
Average quality criterion	-55.86	-52.82	-63.26

Figure 5.5: Tables summarizing the performances of the coordination methods for different letters flow configurations.

5.6 Summary

In order to solve the coordination problem we have been inspired by the functioning of the action selection device of natural control systems. We proposed a computational model based on restless bandits indexes that implements such a device and showed that its performances overcome those of an existing method. However we have used the behavior selection scheme without interruption in our implementation, because so far, we do not have a clear idea about how interruption should work. We think that this issue is of great importance and we will investigate it in our future work.

[5]Personal communication from John Tsitsiklis

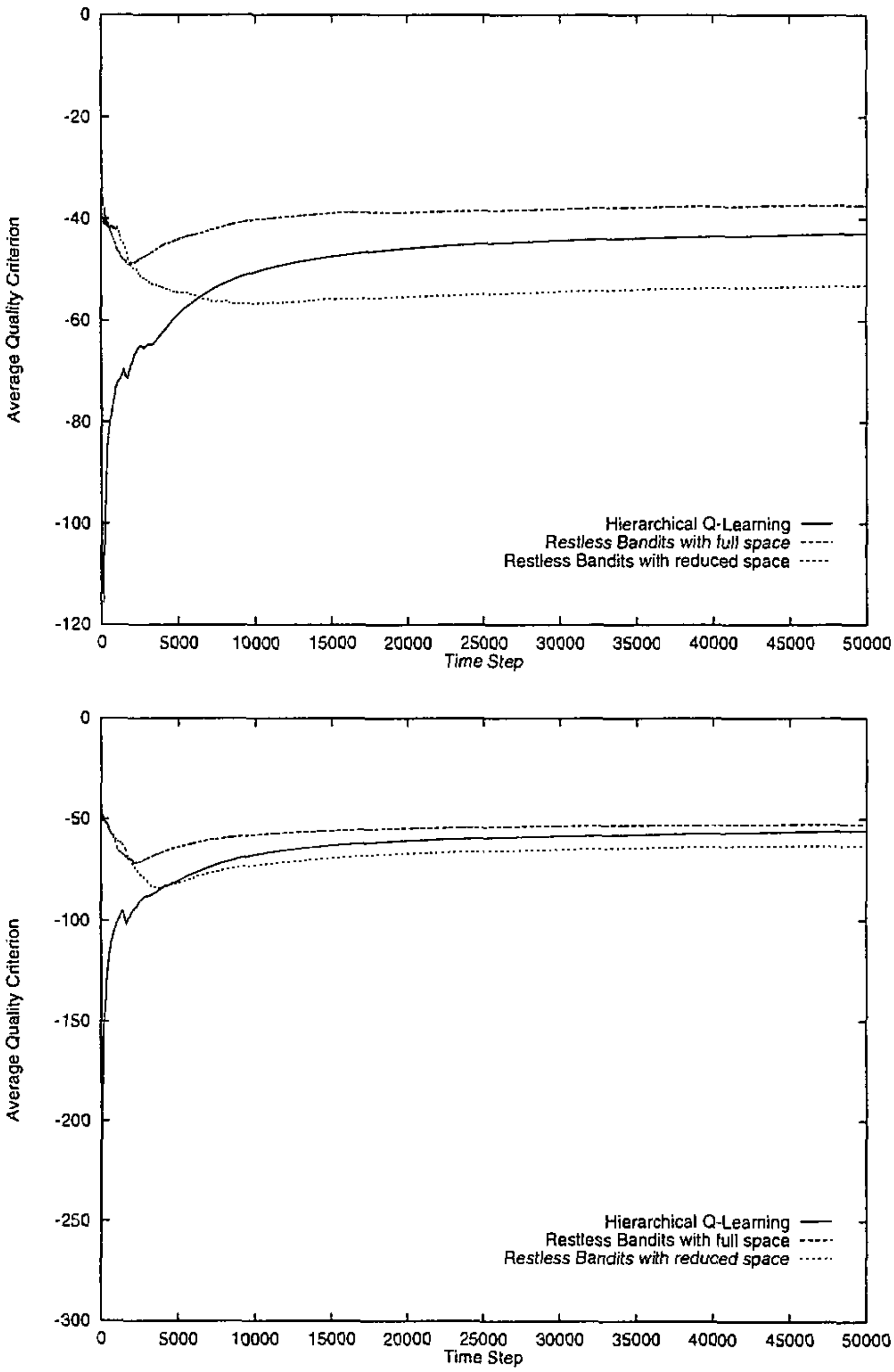

Figure 5.6: Average of the quality criterion as a function of decision steps. The top graph concerns the periodic letters flow and the bottom graph the Poisson distribution letters flow.

Chapter 6

Conclusion

6.1 Summary of contributions

The work presented in this thesis was motivated by the need to solve complex problems using embedded agents learning by reinforcement. We identified and analyzed the reasons that make standard reinforcement learning methods impractical in complex domains and proposed some mechanisms to scale up these approaches. Our contributions are summarized as follows.

We set up a new design methodology whose aim is to systemize the agent's design process (Faihe and Müller 1997). It provides a conceptual framework to design hierarchical control architectures for embedded autonomous agents. The objectives of each stage of the methodology were clearly defined and the distinction was made between what the agent has to learn and what has to be given *a priori* by the designer.

Assuming that the solution to the problem corresponds to a particular pattern of interaction between the agent and the environment, we established the relationship between solving a problem and generating a behavior. Then we proposed a way of formally specifying a behavior. To do so we used a quality criterion, composed of an objective function and a set of constraints. The desired behavior is the one generating a trajectory (in the interaction space) that optimizes the objective function without violating the constraints. In addition to being both a formal and natural means of defining a behavior, the proposed method allows us to derive the reinforcement function (as a progress estimator), to learn the behavior and to have a good basis for the decomposition process.

A graphical approach was proposed to perform the problem's decomposition (or behavior's decomposition because to each problem corresponds a set of behaviors that solves it). Although this technique is still partly reliant on the designer's intuition and experience, it allows to discover sub-behaviors that would not be identified otherwise.

Concerning the coordination problem, we reviewed the features present in the behavior's selection mechanism of natural systems and highly desirable in artificial systems. We proposed a coordination method based on restless bandits indexes (Faihe and Müller 1998). It extends and generalizes W-learning, is completely distributed and has been shown to be more powerful than Hierarchical Q-learning for the postman robot task.

The feasibility of the methodology as well as the performances of the methods were demonstrated through the postman robot problem, which is a non-trivial problem. In addition we developed and implemented a three-level architecture, which is rarely found in the reinforcement learning area.

6.2 Practical Issues

The implementation of the postman robot architecture was not straightforward and sometimes resulted in agents that fail to converge to a satisfactory solution. The main difficulty was finding a good tuning of the parameters, which are the learning rate α, the eligibility trace factor λ, the discount factor γ and the number of exploration steps N_{exp}. Unfortunately there is no scientific method to tune such parameters so they are chosen according to one's own experience and experiments as well as those reported by other researchers.

We noticed that α and λ are closely linked and that the evolution of one of them affects the value of the other. A bad setting of these parameters results either in a slow convergence or in a complete failure of the learning process. We decided to measure the performances of the agent (i.e. the average value of the quality criterion after 50000 decision steps) for several values of α and λ. The best results were obtained for $\alpha = 0.5$ and $\lambda = 2.0$, which are the values used during our experiments for all the architectures.

The number of exploration steps was easy to find. Starting with a small value of N_{exp}

(200 steps), we increased it progressively until 8000 steps and reported the agent's performance. We noticed that the performance improves while N_{exp} increases, then stabilizes between 1500 and 4000 steps, and deteriorates thereafter. In effect, if the value of N_{exp} is too low then the agent will be unable to find a good policy (due to the lack of search) and, on the other hand a high value will prevent the agent consolidating its knowledge because of the random perturbations.

For the discount factor one may wonder whether to discount ($\gamma < 1$) or not ($\gamma = 1$). Discounting is useful for any task that is learned in trials. The navigation tasks, for example, are suitable to be learned with discounting because solutions that allow the agent to reach the goal in very few steps are preferred. The scheduling task (coordination of the navigation's behaviors) is a continuous task. Therefore a natural and logical optimality criteria would be the average reward received over time. General results for online learning using such a criterion are currently under progress (Mahadevan 1994). However we obtained fair performances with $\gamma = 0.99$.

Another difficulty we had to face concerns the stability of neural networks. It was impossible to get a stable network with a linear output unit, even with a very low learning rate (order of magnitude of 10^{-3}). For this reason we used networks with non-linear output units. Nevertheless we were constrained to scale the reinforcement value between -0.1 and 0.1 to avoid large updates, which may make units blow up.

6.3 Future work

Further research that can be carried out in the direction of the work presented in this dissertation is twofold. It may concern the extension of the methodology or the improvement of the proposed methods.

One possible way of extending the methodology would be to automate the processes, which require extensive human intervention. Such processes are the decomposition of a behavior into sub-behaviors and the design of sensory-motor loops. We do believe that animals, which learn by reinforcement (such as birds learning to fly), were born with all the necessary structures to achieve such a learning. These structures are genetically transmitted and evolved through several generations to fit in their environment. In our framework we are interested in finding a hierarchy of sub-behaviors with, for each of them, the reinforcement function as well as the sets of relevant perceptions and commands. It is possible to do so using genetic algorithms but we still need to find out a means of describing the representation and the dynamics of the agent's internal structure.

The behavior specification method we proposed may have some drawbacks in case the agent does not have the ability to sense features that allow frequent updates of the quality criterion. This problem, particularly arising in robotics, may make the learning system fail because of the lack of immediate reinforcements. A common way of adding immediate reinforcements is to provide the agent with advice. Advice comes from a teacher's visual evaluation of the agent's performances and may completely bias the learning procedure or make the agent exhibit unexpected behaviors because it is difficult to put oneself in the agent's shoes. Therefore an interesting issue would be to find a way of carefully integrating such advice in the quality criterion.

A mathematical approach to perform the decomposition is highly desirable in the sense that it will allow us to understand this process and to automate it.

The coordination problem has to be investigated within a theoretical framework. The most suitable one is proposed by Sutton et al. (1998). It consists in solving MDPs using macro-actions and involves temporal and behavior abstractions as well as macro-actions' interruption. Interesting directions for investigation concern the state abstraction and the implementation of a behavior's persistence. In the latter direction, an important issue would be to identify states where it is worth interrupting macro-actions in order to avoid updating and making a new decision in each state of a macro-action's trajectory.

Finally, an intensive application of the HPS methodology to different problems in different areas would help to find out its weaknesses and overcome them.

6.4 Epilogue

The work presented in this thesis takes place within the general context of learning and development in artificial creatures. The long-term objective is to find mechanisms that allow animats to incrementally develop their intelligence in a constructivist manner. It means that they have to discover and develop by themselves the building blocks that will be used to build more and more complex skills. The main rule is that they can only learn what is close to what they already know. We had the opportunity to verify this rule during the implementation of the hierarchical architecture (section 4.5.2). We were unable to obtain a stable strategy using what Dorigo and Colombetti (1998) called *holistic learning*, that is, learning from scratch all the behaviors of each level at the same time, even when learning was delayed between the levels (by increasing the number of exploration steps or reducing the learning rate of upper behaviors). This is why we adopted a modular learning approach.

Bibliography

Barraquand, J. and J. C. Latombe (1991). Robot motion planning: A distributed representation approach. *The International Journal of Robotics Reasearch 10*(6), 628–649.

Barto, A., R. Sutton, and C. Watkins (1990). Learning and sequential decision making. In *Learning and sequential decision making*, M.Gabriel and J.W. Moore, editors, The MIT Press.

Barto, A. G., S. J. Bradtke, and S. P. Singh (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence 72(1-2)*, 81–138.

Barto, A. G. and S. P. Singh (1990). On the computational economics of reinforcement learning. In D. S. Touretzky (Ed.), *Connectionist Models: Proceedings of the 1990 Summer School.* Morgan Kaufmann.

Benbrahim, H. and J. A. Franklin (1997). Biped dynamic walking using reinforcement learning. *Robotics and Autonomous Systems 22*, 284–302.

Bradtke, S. J. and M. O. Duff (1995). Reinforcement learning methods for continuous-time markov decision problems. In *Advances in Neural Information Processing Systems 7.* MIT Press.

Braitenberg, V. (1984). *Vehicles. Experiments In Synthetic Psychology.* MIT Press.

Cichosz, P. (1995). Truncating temporal differences: On the efficient implementation of td(λ) for reinforcement learning. *Journal of Artificial Intelligence Research 2*, 287–318.

Colombetti, M., M. Dorigo, and G. Borghi (1996). Behavior analysis and design - a methodology for behavior engineering. *IEEE Transactions on Systems, Man and*

Cybernetics 26.

Crites, R. H. (1996). *Large-Scale Dynamic Optimization using Teams of Reinforcement Learning Agents.* Ph. D. thesis, University of Massachusetts.

Dayan, P. and G. E. Hinton (1993). Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5.*

Dean, T. and S.-H. Lin (1995). Decomposition techniques for planning in stochastic domains. Technical Report CS-95-10, Brown University.

Dietterich, T. G. (1997). Hierarchical reinforcement learning with the MAXQ value function decomposition. Technical report, Oregon State University.

Dorigo, M. and M. Colombetti (1998). *Robot Shaping: An Experiment in Behavior Engineering.* MIT Press/Bradfort Books.

Faihe, Y. and J.-P. Müller (1997). Behavior analysis and design: Towards a methodology. In A. Birk and J. Demiris (Eds.), *Proceedings of the Sixth European Workshop on Learning Robots (EWLR6)*, Lecture Notes in Artificial Intelligence. Springer-Verlag.

Faihe, Y. and J.-P. Müller (1998). Behaviors coordination using restless bandits allocation indexes. In *Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior (SAB98).*

Gittins, J. C. (1989). *Multi-armed Bandit Allocation Indices.* Willey.

Hauskrecht, M., N. Meuleau, C. Boutilier, L. P. Kaelbling, and T. Dean (1998). Hierarchical solution of markov decision processes using macro-actions. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI98).*

Humphrys, M. (1997). *Action Selection methods using Reinforcement Learning.* Ph. D. thesis, University of Cambridge.

Kaelbling, L. P. (1993a). Hierarchical learning in stochastic domains: Preliminary results. In M. Kaufmann (Ed.), *Proceedings of the Tenth International Conference on Machine Learning.*

Kaelbling, L. P. (1993b). *Learning in Embedded Systems.* MIT Press.

Kaelbling, L. P., M. L. Littman, and A. W. Moore (1996). Reinforcement learning: a survey. *Journal of Artificial Intelligence Research 4*.

Kalmàr, Z., C. Szepesvàri, and A. Lörincz (1998). Module-based reinforcement learning: Experiments with a real robot. *Machine Learning*. In press.

Khatib, O. (1986). Real time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Reasearch 5*(1), 90–98.

Kröse, B. J. A. and J. W. M. Van Dam (1993). Learning to avoid collisions: a reinforcement learning paradigm for mobile robot navigation. In *Proceedings of International Symposium on Artificial Intelligence in Real-Time Control (IFAC)*.

Lin, L. J. (1992). Reinforcement learning with hidden state. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*.

Lin, L. J. (1993). Hierarchical learning of robot skills by reinforcement. In *Proceedings of the IEEE International Conference on Neural Networks*.

Mahadevan, S. (1994). To discount or not to discount in reinforcement learning: A case study comparing R-learning and Q-learning. In *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann.

Mahadevan, S. (1996). Optimality criteria in reinforcement learning. Presented at the AAAI Fall Symposium on Learning Complex Behaviors in Adaptive Intelligent Systems.

Mahadevan, S. and J. Connell (1992). Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence 55*, 311–365.

Mahadevan, S., N. Marchalleck, D. T., and G. Abhijit (1997). Self improving factory simulation using continuous-time reinforcement learning. In M. Kaufmann (Ed.), *Proceedings of the Fourteenth International Conference on Machine Learning*.

Martin, M. M. (1998). *Reinforcement Learning for Embedded Agents Facing Complex Tasks*. Ph. D. thesis, Universitat Politecnica de Catalunya.

Mataric, M. J. (1994). Reward functions for accelerated learning. In *Proceedings of the Eleventh International Conference on Machine Learning.* Morgan Kaufmann.

McCallum, A. (1996). *Reinforcement Learning with Selective Perception and Hidden State.* Ph. D. thesis, University of Rochester.

McFarland, D. (1981). *Animal Behaviour.* Longman.

Meuleau, N. and P. Bourgine (1998). Exploration of multi-state environments: Local measures and back-propagation of uncertainty. *Machine Learning.* In press.

Millàn, J. d. R. (1996). Rapid, safe and incremental learning of navigation strategies. *IEEE Transactions on Systems, Man and Cybernetics 26.*

Minoux, M. (1986). *Mathematical Programming.* John Wiley and Son.

Parr, R. (1998). Flexible decomposition algorithms for weakly coupled markov decision problems. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI98).*

Peng, J. and R. J. Williams (1996). Incremental multi-step Q-learning. *Machine Learning 22*, 283–290.

Pfeifer, R. (1996). Building "fungus eaters": Design principles of autonomous agents. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB96).*

Pfeifer, R. and C. Scheier (1998). *Introduction to "New Artificial Intelligence".* MIT Press. Book manuscript under review.

Pomerleau, D. A. (1991). Efficient training of artificial neural networks for autonomous navigatioin. *Neural Computation 3*(1), 88–97.

Prescott, T. J. and J. E. Mayhew (1992). Obstacle avoidance through reinforcement learning. In *Advances in neural information processing systems 4*, pp. 523–530. Morgan Kaufmann.

Prescott, T. J., P. Redgrave, and G. Kevin (1999). Layered control architectures in robots and vertebrates. *Adaptive Behavior.* To appear.

Redgrave, P., T. J. Prescott, and G. Kevin (1999). The basal ganglian: A vertebrate solution to the selection problem? *Neuroscience*. To appear.

Rummery, G. A. (1995). *Problem Solving With Reinforcement Learning*. Ph. D. thesis, University of Cambridge.

Rummery, G. A. and M. Niranjan (1994). On-line Q-learning using connexionist systems. Technical Report CUED/F-INFEG/TR66, Cambridge University.

Santamaria, J. C., R. S. Sutton, and A. Ram (1997). Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior 6*(2), 163–217.

Simmons, R., R. Goodwin, K. Z. Haigh, S. Koenig, and J. O'Sullivan (1997). A modular architecture for office delivery robots. In *Proceedings of the First International Conference on Autonomous Agents*. ACM Press.

Singh, S. P. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning 8*(3/4), 323–339.

Singh, S. P. and D. Bertsekas (1997). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems*. MIT Press.

Singh, S. P. and R. S. Sutton (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning 22*, 123–158.

Stephens, D. W. and J. R. Krebs (1986). *Foraging Theory*. Princeton University Press.

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

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

Sutton, R. S., D. Precup, S. Singh, and B. Ravindran (1998). Improved switching among temporally abstract actions. In *Advances in Neural Information Processing Systems 11*. MIT Press.

Tesauro, G. (1995). Temporal difference learning and td-gammon. *Communication of the ACM 38*, 58–68.

Tham, C. L. (1995). Reinforcement learning of multiple tasks using hierarchical CMAC architecture. *Robotics and Autonomous Systems 15*(4), 247–274.

Thrun, S. (1992). Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, Carnegie Mellon University.

Tyrell, T. (1993). The use of hierachies for action selection. In *Proceedings of the Second International Conference on the Simulation of Adaptive Behaviour (SAB92).*

Watkins, C. (1989). *Learning from delayed rewards.* Ph. D. thesis, University of Cambridge.

Weber, R. R. and G. Weiss (1989). On an index policy for restless bandits. *Journal of Applied Probability 27.*

Whitehead, S., J. Karlsson, and J. Tenenberg (1993). *Learning Multiple Goal Behavior via Task Decomposition and Dynamic Policy Merging.* Kluwer Academic Publishers.

Whittle, P. (1988). Restless bandits : Activity allocation in a changing world. *Journal of Applied Probability 25.*

Wiering, M. and J. Schmidhuber (1998). Fast online Q(λ). *Machine Learning.* In press.

Wilson, S. W. (1996). Explore/exploit strategies in autonomous learning. In J.-A. M. Pattie Maes and S. Wilson (Eds.), *From animals to animats 4: Proceedings of the Fourth International Conference on the Simulation of Adaptive Behaviour.* MIT Press.

Wyatt, J. (1997). *Exploration and Inference in Learning from Reinforcement.* Ph. D. thesis, University of Edinburgh.

Wyatt, J., J. Hoar, and G. Hayes (1998). Design, analysis and comparison of robot learners. *Special issue on Scientific Methods in Mobile Robotics: The New Wave, Robotics and autonomous Systems 24*(1-2).

Zhang, W. and T. G. Dietterich (1995). A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann.