UNIVERSITAT POLITÈCNICA DE CATALUNYA
DEPARTAMENT DE LLENGUATGES I SISTEMES INFORMÀTICS
PROGRAMA DE DOCTORAT EN INTEL·LIGÈNCIA ARTIFICIAL

TESI DOCTORAL

Reinforcement learning for embedded agents facing complex tasks

April 1998

Memòria presentada per Mario Martín Muñoz per a optar al títol de Doctor en Informàtica
Director: Ulises Cortés García

UPC

Departament de Llenguatges i Sistemes Informàtics

Universitat Politècnica de Catalunya
Barcelona, Catalunya

CERTIFICATE OF APPROVAL

Ph.D. Thesis

This is to certify that the Ph.D. Thesis of

Mario Martin

has been approved by
the Examining Committee on June 2, 1998
as satisfactory for the thesis requirement
for the Ph.D. in Computer Science degree

Examining Committee :

President: Ramón López de Mántaras, Ph.D.

Secretary: Miquel Sanchez Marré, Ph.D.

Member: Leslie Pack Kaelbling, Ph.D.

Member: Jürgen Schmidhuber, Ph.D.

Member: Enric Celaya, Ph.D.

Abstract

The aim of this work is to develop reinforcement learning mechanisms to allow embedded agents to deal with complex tasks.

A complex task can be informally described as one that shows a long delayed reinforcement, that deals with a limited perception of the environment, and that is defined not only by a specific problem or setting of the environment but for a general class of problems.

To learn this kind of tasks, we propose on one hand, a new algorithm for embedded agents, called CANDID, focused on learning *general*, *reliable* behaviors with an *acceptable performance* that can work with a moderate *perceptual aliasing*. This new algorithm is characterized by the following features: it learns from experiences in a trial-based way, learning is based in the final reinforcement obtained for each trial, it does not learn optimal policies but acceptable ones (defining a maximum reasonable cost for all experiences), it does not use exploration techniques, and finally, it presents a new forgetting mechanism that works as a dynamic α parameter.

In order to solve the perceptual aliasing problem, we propose a new mechanism that we call `Active Memorization`. The main difference between this approach and other approaches previously developed to solve this problem, consists in the selective process used to store information about the history of the solving process. This mechanism defines a set of actions that are able to manage the memory resources of the agent. In this way, the agent learns to solve the task by learning also to memorize the relevant information about the environment that could be useful in the future in order to disambiguate the aliased states. The performed experiments show that CANDID working with `Active Memorization` is able to learn a set of tasks with perceptual aliasing in a moderate number of trials.

In order to solve the long delayed reinforcement problem, we propose in this work a hierarchical approach to learn in an incremental way intermediate behaviors useful for the solving the initial task. Experiments show that the hierarchical mechanism we are proposing becomes an efficient approach suited to CANDID and `Active Memorization` that allows learning general behaviors with a long delayed reinforcement.

Finally, we show how the proposed mechanisms can be fruitfully applied to robotic tasks in continuous environments with noise.

Acknowledgments

Carrying out this thesis required the support and help of a lot of people that, first of all, I would like to mention here.

First and foremost, I would thank Ulises Cortés for pushing me to carry out this work, for his dedication, for his criticisms, and for his efficient guide during all these years.

I'm grateful to Leslie Kaelbling, Jürgen Schmidhuber and Marco Wiering for reviewing a previous version of this work and for proposing some modifications that have improved a lot this document. I would also thank to the referees that have evaluated parts of this work in several conferences, for their constructive criticisms.

I would thank Ramón Sangüesa, colleague and deskmate, for his friendship and for trying to remove some of the infinite language mistakes of this work.

I am in debt with Neus Català for implementing the `Candid-Cooker`, but especially for her patience in the critical moments, her loving care, her moral support, and so many other things that she knows and that would fill up an entire book.

I am grateful to the LSI department of the Technical University of Catalonia for reducing my teaching load in some critical times, and to the technical staff of the same department that have made my life easier these years.

Finally, I would thank my friends and, especially, my family that have given me their support throughout the time spent in this work.

Contents

CHAPTER 1 Introduction **1**
1.1 Motivation of this work 1
1.2 Background 2
1.3 Roadmap 3

CHAPTER 2 Learning to solve complex tasks with embedded agents **5**
2.1 Introduction 5
2.2 Features of complex tasks 6
2.2.1 Large length of the solution 7
2.2.2 Large number of situations 8
2.2.3 Limited perception 9
2.2.4 General behaviors 9
2.3 Our proposal 10
2.4 Features of a desirable learning algorithm 11
2.4.1 Learning general behaviors 11
2.4.2 Learning behaviors of acceptable performance 12
2.4.3 Learning reliable behaviors 13
2.4.4 Ability to deal with ambiguous perceptions 14
2.5 Other issues 14
2.6 Summary 17

CHAPTER 3 The Learning Algorithm: CANDID **19**
3.1 A new learning mechanism 19
3.2 An example application 24
3.2.1 Environment: blocks world 24
3.2.2 The task of searching for a block 26
3.2.3 Performance 28
3.3 Parameters for applying CANDID to a task 31
3.4 Variations of the CANDID algorithm 36
3.4.1 Using variable punishment 36
3.4.2 Heuristics 38
3.5 Comparison with other works 41

3.5.1 Theoretic comparison: some observations 42
3.5.2 Empirical comparison 46
3.6 Conclusions 50

CHAPTER 4 Active Memorization 55
4.1 Introduction 55
4.2 Previous Works 56
4.2.1 "Ignoring" the problem 56
4.2.2 Avoiding ambiguities: Animate Perception 57
4.2.3 Disambiguate states by learning a world model 58
4.2.4 Belief vectors 59
4.2.5 Automatic Programming 60
4.2.6 Hierarchical approach 61
4.2.7 State generation approaches 61
4.2.8 Memory approaches 62
4.3 Active Memorization 63
4.3.1 Antecedents 64
4.3.2 New Active Memorization Models 68
4.3.3 Comparison 72
4.4 Test Domains 77
4.4.1 Limited perception I 77
4.4.2 Limited perception II 85
4.5 Conclusions 90

CHAPTER 5 Scaling up: Incrementality and Development 93
5.1 Introduction 93
5.1.1 Teaching 93
5.1.2 Partial modeling 95
5.1.3 Guided exploration 96
5.1.4 Incremental learning 97
5.2 Incrementality on the size of the task 98
5.3 Incrementality on the kind of the task 103
5.3.1 Previous work 103
5.4 Our approach: hierarchical development 110
5.5 Experiments: The fruitcake problem 114
5.6 Conclusions and future work 132

CHAPTER 6 Application to a realistic robot simulator 137
6.1 Introduction 137
6.2 Description of the Khepera robot and simulator 138
6.3 Description of the task 140
6.4 Adaptation of Candid to robotic tasks 142
6.5 Experiments 144

6.5.1 Experiments and results 147
6.6 Incremental learning 153
6.6.1 A new incremental approach 153
6.6.2 Description of the *solve corners task* 155
6.6.3 Results of incremental learning for robots `R1` and `R2` 156
6.7 Improving reliability: Extra-learning 158
6.8 The role of noise 161
6.9 Conclusions 163

CHAPTER 7 Conclusions **165**
7.1 Contributions 165
7.2 Future work 170
7.2.1 Candid 171
7.2.2 Active Memorization 172
7.2.3 Hierarchical development 172

APPENDICES **175**

Appendix A Properties of the weighting procedure **177**
A.1 Introduction 177
A.2 First Statement 177
A.3 Second invariance 180

Appendix B Variable punishment **183**
B.1 Statement 183
B.2 Proof 185

Appendix C Exhaustivity and non redundancy in Candid **189**
C.1 Statement 189
C.2 Proof 189
C.2.1 Building the search tree 190
C.2.2 Correspondence between chains of actions and branches of the search tree 191
C.2.3 Depth-first search and punishment in Candid 192

Appendix D Dynamic α **199**
D.1 Statement 199
D.2 Proof 200

Appendix E Learned policies **203**
E.1 Introduction 203
E.2 McCallum's mazes 203
E.3 Littman, Kaelbling and Cassandra's mazes 207
E.4 4x4 Grid 209

E.5 39x38 maze 209
E.6 *Going out of a room* using 3 virtual sensors composed from 6 real ones 210
E.7 *Going out of a room* with 6 sensors 211

Bibliography **212**

List of Figures

3.1 A standard setting of the environment. 26
3.2 Different initial conditions for the general task of *searching for a block*. In these settings, the goal block is the black one. 27
3.3 Initial performance of the new learning algorithm. 29
3.4 Performance of CANDID for different values of the parameter N. . 30
3.5 Application of the general learned behavior for two different settings. In these settings, the goal block was the black one. 32
3.6 There is no optimal action. 43
3.7 Comparison of the CANDID algorithm with `Sarsa(1)`, `backward Q-learning` and `Q-learning`. 49

4.1 Increase of the number of actions for each approach as the number of register memory grows. The number of original actions is set to 4. 75
4.2 Increase of number of policies for each approach as the number of register memory grows. The number of original actions is set to 4 and the number of perceptual states 16. Policies scale is logarithmic. 76
4.3 McCallum's mazes (a,b,c,d) [123], (e) [125]. 78
4.4 57-states and 89-states [97]. The numbers inside the cells are perceptible by the agent. 82
4.5 4x4 maze [33]. 86
4.6 38x39 Grid [208]. 88

5.1 Incremental problem [155]. 99
5.2 A standard setting of the environment. 115
5.3 Hierarchy of behaviors for the *fruitcake problem*. Intermediate behaviors to solve the *fruitcake problem* are encircled. Arrows represent the used-as-action-to-build-behavior relationship. 120
5.4 Learning curve for the task of `searching for a block`. 121
5.5 Learning curve for the task of `moving the hand to the visual focus`. 123
5.6 Learning curve for the task of `removing the top of a stack`. . 124
5.7 Learning curve for the task of `grasping a block`. 126

5.8 Learning curve for the task `moving a block where the visual mark lays`. . . . 128
5.9 Modified environment for solving the fruitcake task 130
5.10 Learning curve for the `Fruitcake-color task`. . . . 131

6.1 Pictures of the real `Khepera` robot and its representation in the simulator (taken from [135]). . . . 138
6.2 Three different rooms used in the learning process of the robot. . 140
6.3 Different actions for the same perception. See explanation on the text. . . . 141
6.4 Trace of a learned behavior in the first room. . . . 151
6.5 Trace of a learned behavior in the second room. . . . 151
6.6 Trace of a learned behavior in the third room. . . . 152
6.7 Two initial conditions and goal locations for each of them in the *solve corners* task. . . . 155

C.1 Example of a transformation of the search tree. The parenthesized labels mark some nodes and actions of the tree with the nomenclature used in the transformation rule. The label inside each node denotes the perception that it represents. . . . 191

List of Tables

4.1 Summary table I. Where s is the number of perceptual states, n is the number of external actions, w is the length of the window measured in bits, c_i is the number of possible values for memory i, r is the number of memory registers, and c_0 is the number of possible values of the oscillator. 73

4.2 Summary table II. Where s is the number of perceptual states, n is the number of external actions, w is the length of the window measured in bits, c_i is the number of possible values for memory i, r is the number of memory registers, c_0 is the number of possible values of the oscillator, and m is the number of memory actions needed in each approach in order to solve the problem (see text). . 74

4.3 Characterization of McCallum's mazes with `Active Memorization I`. 79

4.4 Results for McCallum's mazes with `Active Memorization I` . . . 81

4.5 Characterization of 57-states and 89-states mazes for `Active Memorization I`. 83

4.6 Results for Littman, *et al.* mazes with `Active Memorization I` . 83

4.7 Comparison of polices for 57-states maze. 84

4.8 Characterization of the 4x4 grid with `Active Memorization II`. . 86

4.9 Results for the 4x4 Grid 87

5.1 Results for Ring mazes with `Active Memorization I` 101

5.2 *Incremental results* for Ring mazes with `Active Memorization I` 102

6.1 Robots used in experiences. 146

6.2 First results with robot `R1` using CANDID and `Active Memorization` with one memory bit. The number of experiences needed to learn the behavior and the percentage of reliability of the learned behaviors are shown for 30 independent runs. 148

6.3 First results with robot `R2` using CANDID and `Active Memorization` with one memory bit. The number of experiences needed to learn the behavior and the percentage of reliability of the learned behaviors are shown for 30 independent runs. 150
6.4 Results with robot `R1` using CANDID, `Active Memorization` with one memory bit, and the *new* incremental approach. The number of experiences needed to learn the behavior are shown for the intermediate task, for the original task, and finally, for the sum of the two tasks. Statistics about the reliability of the learned behaviors for the original task in 30 independent runs are also shown. 157
6.5 Results with robot `R2` using CANDID, `Active Memorization` with one memory bit, and the *new* incremental approach. The number of experiences needed to learn the behavior are shown for the intermediate task, for the original task, and finally, for the sum of the two tasks. Statistics about the reliability of the learned behaviors for the original task in 30 independent runs are also shown. 158
6.6 Results obtained with robot `R1` using CANDID, `Active Memorization` with one memory bit. There are shown the number of experiences needed to learn the behavior in the common way (the results shown in table 6.2), the number of experiences needed in the extra-learning, and finally, the accumulated number of experiences for the two phases. Statistics about the reliability of the learned behaviors for the original task in 30 independent runs are also shown. 160
6.7 Number of experiences needed for learning the behavior in the incremental way and percentage of reliability of the learned behaviors for 30 independent runs *without noise*. Compare these results with the ones obtained with noise in table 6.4. 161
6.8 Summary of the experiments performed in this chapter. 164

E.1 Best learned policy for the *a* McCallum maze 203
E.2 Best learned policy for the *a2* McCallum maze 204
E.3 Best learned policy for the *b* McCallum maze 204
E.4 Best learned policy for the *c* McCallum maze 205
E.5 Best learned policy for the *d* McCallum maze 205
E.6 Best learned policy for the *e* McCallum maze 206
E.7 Best learned policy for the *e2* McCallum maze 206
E.8 Best learned policy for the *57-states* maze 207
E.9 Best learned policy for the *89-states* maze 208
E.10 Best learned policy for the *4x4 Grid* 209

E.11 Best learned policy for the *39x38 maze* 209
E.12 Best behavior found by Kehepera using the CANDID algorithm with the information of 6 sensors grouped in 3 virtual sensors in the *going out of a room* task. 210
E.13 Best behavior found by Kehepera using the CANDID algorithm with 6 sensors in the *going out of a room* task. 211

Chapter 1

Introduction

1.1 Motivation of this work

Intelligence, as the ability to solve problems, must be inevitably embedded in an environment. We consider essential to face the artificial intelligence issue from the point of view of its grounding in the environment.

This thesis is about *embedded agents* [83], that is, agents totally integrated in the environment, designed with a set of specific sensors and actions adequate for the environment, and that show a quick response to the environment.

There has been a lot of research in embedded agents, sometimes known as *reactive systems*, *situated agents* [35] [3], *routine-guided agents* [2], *enactive agents* [204] or *behavior based agents* [30]. All of these terms are not synonymous because they are focused on different issues of agent design, but all of them share most of their assumptions.

Embedded agents are very effective in a lot of different tasks, standing out in robotic applications. Nevertheless, most of the applications made with embedded agents are usually programmed by human experts.

Some work has been done in learning methods for embedded agents, mostly based in the idea of learning from a reinforcement signal. This is the most suitable kind of learning for embedded agents because the agent discovers the adequate behavior from its own interaction with the environment. Reinforcement learning consists in allowing the agent to operate in the environment by trying to solve a proposed problem. The behavior performed by the agent is evaluated with a reinforcement function. The evaluation obtained is given to the agent which changes its behavior depending on the evaluation (that is, it learns).

Nevertheless, learning is usually limited to find, with complete information about the environment and not a very long delayed reinforcement, an optimum behavior. Unfortunately, not all tasks fulfill these conditions. The main motivation of this work is to allow embedded agents to learn the behaviors necessary for solving tasks that do not fulfill those conditions, that is, the learning of complex tasks.

1.2 Background

In this work, we assume that the reader is familiar with the classical reinforcement learning methods, for instance `TD(λ)` [180] and `Q-learning` [205]. Nevertheless, we give here a brief glossary containing some terms of reinforcement learning used in this work:

Task: A description of the class of problems the agent has to solve by using the learned behavior.

Problem: One particular instance of the task the agent has to solve.

Trial: The execution performed by an agent for a proposed problem.

Experience: As in the trial definition, the execution performed by an agent for a proposed problem, but also considering the reinforcement the agent obtains for the execution performed.

Reinforcement signal: The reinforcement signal is used to evaluate the behavior of the agent. It can be immediate or delayed. Immediate reinforcement consists in returning to the agent an evaluation after the execution of each action. Delayed reinforcement consists in returning to the agent a global evaluation of the behavior after the whole execution of a trial. In this work we will deal with delayed reinforcement.

Optimum behavior: The best behavior in terms of a function that usually measures the total amount of reinforcement obtained in the trial execution. This function usually includes a parameter γ (discounting parameter) that determines the balance between the desirable near-in-time reinforcement to obtain and the desirable distant-in-time reinforcement to obtain. In this way, the function returns, when $\gamma < 1$, a maximum evaluation for behaviors that obtain a maximum discounted reinforcement in a minimum of steps.

Policy: The current behavior in the learning process which describes a mapping from situations perceived to actions to execute. It is defined according to a criterion that varies depending on the learning mechanism used. The more usual criterion is to choose the action for each situation that will return a maximum discounted reinforcement.

Markovian property: A task fulfills the Markovian property in an environment when the transition probabilities from one state to another one *only* depends on the action executed. This property allows to predict the next state with independence of past actions executed and/or states visited. Most reinforcement learning algorithms, in order to decide the action to be executed, assume (directly or indirectly) that the Markovian property holds in the task faced. Those algorithms propose the action to execute depending on the probability of leading to the rewarding states that, in turn, depends on those transition probabilities. Unfortunately the Markovian property is not always fulfilled. Some tasks (when using those algorithms) that *do not fulfill this property* can be detected when the procedure that determines the best action to be executed for the current perceptual state, can be improved with additional information about the past actions taken and/or past states perceived.

1.3 Roadmap

In order to scale up reinforcement learning in embedded agents to solve complex tasks, we propose in this work a new set of methodologies, mechanisms and algorithms.

First of all, in chapter 2 we analyze the features that characterize complex tasks for reinforcement learning with embedded agents. From this analysis, we outline a new methodology to allow learning complex tasks. To implement this methodology, a new learning algorithm that fulfills a set of properties (also defined there) is required.

In chapter 3, we present a new learning algorithm called CANDID that we have developed according to the requirements described in chapter 2. The new algorithm is presented, analyzed, compared with other known reinforcement algorithms and put to test in a first task.

In chapter 4, a new mechanism to solve the inability for learning caused by a limited perception of the environment, is presented. The new mechanism, called `Active Memorization`, is applied to the CANDID algorithm and is compared with other known approaches. Finally, it is tested in a set of benchmark environments.

In chapter 5, a methodology for solving tasks that imply a long delayed reinforcement under perceptual limitations, is presented. The methodology, called `hierarchical development`, which is adequate to CANDID and `Active Memorization`, is compared with other approaches to that problem. Finally, it is tested in a complex task that presents a long delayed reinforcement.

In chapter 6, the applicability of the approach proposed in this work to robotic applications is discused. This kind of applications is interesting because it presents some features that sometimes are obviated in other works. Two of these features are *noise* and *working in a continuous environment*. Another constraint needed to fulfill in robotic applications is the need for *learning from a low number of experiences*. Minor modifications have to be made in the learning mechanisms proposed in this thesis to fulfill these conditions. The resulting learning mechanisms are tested in a representative robotic task.

Finally, in chapter 7 we make an evaluation of the tools proposed in this work, remarking the new contributions to the area and discussing some ideas that could be useful to develop in the near future.

Chapter 2

Learning to solve complex tasks with embedded agents

"Reality is just a convenient measure of complexity."

Fortunes software,
Alvy Ray Smith.

2.1 Introduction

In this chapter, the features that characterize complex tasks for reinforcement learning with embedded agents and the approach that we will follow to learn this kind of tasks, are discused.

In this first section, we make a sketch of the framework within we set our research. In the next section, we define what we consider as complex tasks and the properties they show. In section 2.3 we outline our approach to allow embedded agents to solve these complex tasks. In section 2.4, the constraints that a basic learning algorithm has to fulfill in our approach are described. Finally, it can be found in the last section of this chapter a discussion about other issues that are usually considered the source of the complexity of a task.

There have been several approaches in the research of learning for embedded agents, usually based on the concept of reinforcement (see [83] for a justification). As we have seen in the previous chapter, reinforcement consists in an internal or external signal that indicates, within a specific situation, whether the executed behavior is appropriate or not. Learning becomes then the looking for a correct mapping between the available set of actions and the set of perceptual states that defines the most rewarding behavior.

The reinforcement signal can be *immediate* or *delayed*. Immediate reinforcement means that the reward is given instantaneously after each action execution, being only informative about the suitability of the chosen action. In the delayed case, the reinforcement signal is given at the end of the trial and is used to evaluate the whole sequence of executed actions.

Obviously it is more complex to learn behaviors using delayed reinforcement, but delayed reinforcement is better suited for embedded agents, where the agent interacts directly with the environment with its own set of sensors and actuators. In fact, more reliable results are obtained by training the agent with a reward signal delivered at the end of the trial, than by training it with a reinforcement signal after each action execution (see section 5.1.1 for a justification of this statement). In the immediate case, in order to deliver the reinforcement signal, some knowledge to determine whether the agent is now nearer to the goal state than before the action execution (which implies an immediate positive reinforcement) or not, is needed. This knowledge is usually not easy to obtain and often, when obtained, very unreliable. Moreover, if this reliable knowledge could be somehow obtained, then it would be easier to program the behavior in the agent than teaching it with immediate reinforcement.

On the other hand, delayed reinforcement does not require to know *how* to reach the goal, but only *when* the goal state has been achieved. In this case, it can be said that the behavior to solve the problem *emerges* or is discovered from the interaction of the agent with its environment using the reinforcement signal and with a minimum of human mediation.

These observations lead us to set our research framework as learning for embedded agents through a delayed reinforcement signal. This is the usual approach for embedded agents and we will see through this thesis references to other works within the same framework. However, we are concerned in developing new mechanisms to scale learning up enough to solve complex tasks.

2.2 Features of complex tasks

When trying to develop mechanisms for learning to solve *complex tasks*, first of all it is necessary to know what a complex task is. In this section we study the properties that make a task a complex one to learn. We consider an instance of a complex task to learn the *fruitcake problem* presented in section 5.5.

One first naive characterization of complex tasks in reinforcement learning for embedded agents can be:

Complex task (first characterization): Any task that cannot be reasonably solved by previous classical reinforcement learning algorithms.

Of course, this is not a valid characterization of a complex task, because it depends on the properties of the existing learning algorithms and not on the properties of the task. In spite of this, it seems intuitively right to think that a complex task will not be learned easily with the classical learning algorithms.

A second and more useful characterization of complex tasks is not be based on the properties of classical algorithms but on the properties of the tasks in which these algorithms have shown a weak performance.

There are a lot of features that have been recognized to be the cause of the complexity of a task when it is learned using classical reinforcement algorithms. After some study, we propose some of them as the more relevant ones. These features will be soon explained in depth. Other issues usually considered also relevant for the complexity of a task will be treated in this work but subordinated to the principal ones. In the last section of this chapter, some pointers to this thesis referring to these other relevant issues are presented in order to facilitate a comparison of our work with others.

Having in mind the above considerations, our second characterization of complex tasks rest as follows:

Complex task (second characterization): Any task that presents some or all of the features that are described in the following sections.

2.2.1 Large length of the solution

The solution of a problem consists of a sequence of actions which receives a reinforcement signal when the entire sequence has been executed (delayed reinforcement). Without any initial knowledge, the embedded agent has only available the *trial-and-error* procedure for solving the problem. As we will see, *it is very hard to learn by trial-and-error how to accomplish goals which imply a long sequence of actions.*

On one hand, the desired solution is one of the whole set of possible action sequences of the same length. The cardinality of this set grows exponentially with this length. Consequently, the probability of finding by *trial-and-error* the right sequence of actions decreases in an exponential way.

On the other hand, the *credit assignment problem* (first recognized in [133]) is increased with the length of the chain of actions to evaluate. That problem consists in giving an adequate evaluation for each action in the sequence from the delayed reinforcement, differentiating between the right chosen and wrongly chosen actions. As the number of actions that compose the solution grows, the task of assigning the right credit to each actions becomes more difficult. Assume for instance, that the credit consists in two possible values that can be assigned to each action (right and wrong). The number of possible credit assignments then also grows exponentially (2^n, where n is the length of the sequence of actions), only one of them being the right assignment.

This is true for most algorithms that could be designed for reinforcement learning. Some usual ways of facing that problem consist in adding more knowledge to the agent to ease the solving of the task, or teaching the agent with a more informative reinforcement function (see sections 5.1.2 and 5.1.1 for a discussion).

Our way (we see it as the most appropriate for embedded agents) of facing the problem of the large number of actions to execute, is to consider that *the set of actions of the agent is not adequate to solve the task, because it does not allow to solve the task in few steps.* In this case, the complexity of a task comes from the inadequacy of the actions the agent can take to solve it.

2.2.2 Large number of situations

Another feature that usually characterizes a complex task is that the number of different possible states of the environment is too large to consider an individual response for each of them.

It is known from the Ginsberg-Chapman debate [69] [34] that, for solving complex tasks with reactive systems, it is necessary in some way to *generalize situations*. Chapman [34] shows that a limited world perception by means of *deictic* sensors [35] could be indicated for reactive agents solving complex tasks.

We derive the term deictic sensors from the concept of *deictic representation* [3], [35]. This kind of representation is opposed to the objective representations we are used to. Deictic representations are those that represent things in terms of their relationship with the agent. For instance, the *ball-in-front-of-me* considered as a knowledge representation element, is a deictic representation that indexically denotes, in terms of the state of the agent, an specific ball. This kind of representation is opposed to the objective representation used in the most applications of reinforcement learning, for instance, those usual representations that report a complete description of the room where the agent is, with its current position in the room [181].

Consequently, *deictic sensors are those that report a deictic representation of the environment.* They report information of the world depending on the position and state of the sensor. The state and position of these sensors are controlled by the motor system of the agent. Most animal's sensors are deictic sensors, for instance, the eyes or the touching sensors of the hand. Examples of this kind of sensors can be found in the experiences described in chapters 3, 4, 5 and 6.

The use of deictic sensors produces a subjective, limited and local perception of the environment. It is subjective because depends on the state of the agent's sensor. It is local and limited because it reports information only about the zone of the environment where the sensor is located or focused.

The use of deictic sensors allows the agent to focus only on the relevant information about the environment, reducing the amount of information available to the agent[1]. These sensors then *reduce also the possible perceptual states* of the agent, producing a very profitable generalization process. This was demonstrated in the solution of the *fruitcake problem* with a reactive system designed by Chapman [34].

[1] This use of deictic sensors is called *animate perception* [14].

Other ways of reducing the number of perceptual states consist in using the classical methods of generalization. See section 2.5 for a discussion about this topic.

The effect produced by deictic sensors by providing the agent with a limited perception relates deictic sensors with the next feature of complex tasks.

2.2.3 Limited perception

In most complex tasks for reinforcement learning with embedded agents (for instance, robotic tasks), the sensory system of the agent does not report *all* the possible information about the environment. In this case, it is said that the *agent's perception is limited*. It could be due to the deliberate use of deictic sensors (as we have seen in the previous section) or, as in most real applications, due to the limited sensory resources of the agent.

Sometimes, the limited perception of the environment becomes insufficient for the agent to obtain the necessary information to solve the task. For instance, assume that the information reported by the sensory system, due to the limited perception of the environment, for two different real states becomes the same perception p. Assume also that for this perception p the action to execute, in any policy that solves the task, depends on the *real* state where the agent is. This perceptual state actually represents two different real states. In this case, usually, past actions and/or perceptions allow to determine which is the real state hidden in the current real perception. Also, the determination of the real state usually improves choosing the action to be executed. We have seen in the previous chapter that when this feature appears, the task defined by the set of perceptions and actions of the agent is *non Markovian*. Unfortunately, most reinforcement learning algorithms do not properly work when the task to be learned does not fulfill the Markovian property.

When the information delivered by the perceptual system is not enough to give an unambiguous description of the current situation, it is said that the agent suffers a *perceptual aliasing* of situations [206]. This problem becomes critical when the disambiguation of some states is essential to solve the task.

If the original task (with all the information about the environment) fulfills the Markovian property, the task (with limited perception) is also called a *partially observable problem* that can be modeled by a *partially observable Markov decision process* (POMDP) [19] or a Markovian problem with *hidden state* [125].

2.2.4 General behaviors

Most tasks solved by reinforcement learning mechanisms consist in achieving an optimal path from any initial state to a one final state in a fixed setting.

Other kind of more complex tasks appears when learning is done not only for just one fixed setting of the problem, but for a repertory of similar settings. Different settings with the same general goal and similar conditions in the environment ought to be solved with the same learned behavior. For instance, finding the way to go out of *any* room is a general task, but to obtain the shortest path to go out of a particular room is not a general task. General behaviors are worth to be learned because they are, by definition, more widely applicable and because they reduce the repertory of behaviors the agent has to maintain. Unfortunately, *learning general behaviors is more difficult because reinforcement is given for solving a particular setting.*

As we will see in section 3.5.1, the requirement of solving general tasks (*together with a limited perception of the environment*) entails that a behavior that returns the optimum solution for *all* problems that compose the general task (the usual definition of an optimum behavior), usually does not exist. Generally, good behaviors solve optimally some of the problems, but not all of them. Thus, either we redefine the concept of optimal behavior for this case or we decide to search not for optimal behaviors but for acceptable behaviors (that is, behaviors that solve all problems within a maximum cost that is defined for each problem of the task).

2.3 Our proposal

In the previous section, the properties that in our view characterize what is a complex task for embedded agents, have been presented. We can divide these properties in two groups: the long delayed reinforcement problem and the other problems.

On one hand, the problem of a long delayed reinforcement, as we have seen in the "large length of a trial" section, can be viewed as the inadequacy to solve the task of the agent's set of actions. The agent is built with a set of primitive actions and there can be some tasks too difficult to be learned just with that set of actions.

Our proposal to solve this problem is inspired in the concept of cognitive development stated by the epistemologist Piaget (see [20] for an interesting review of Piaget's work from the point of view of Artificial Intelligence). He claimed that intelligence develops though experiences in steps, from one cognitive stage to a higher one. In his theory, learning is a process in which experiences are assimilated to previously learned structures.

In practice, our approach consists in making the agent learn more adequate actions for solving the complex task. These new actions must allow the agent to solve the task in a few number of steps, disappearing in this way the long delayed reinforcement problem.

To use these new actions, the agent previously has to learn them. They will be learned by proposing to the agent a set of tasks. For each task, a new behavior is learned. These learned behaviors can be considered as new actions that increase the repertory of actions of the agent. The increase in the repertory of actions allows the agent to face more complex problems, representing a cognitive development towards a higher level.

Thus, to solve the long delayed reinforcement problem, a *hierarchical learning* process is proposed (details of this approach, and the adequacy of other different approaches to the problem are discussed in chapter 5). The hierarchical learning we propose describes a developmental process in the agent's abilities over time, and therefore we name it `hierarchical development`.

On the other hand, in reference to the second group of properties of complex tasks, our approach will consist in creating a new mechanism able to learn general behaviors with deictic sensors and limited perception. This new mechanism must be also adequate for allowing the hierarchical development introduced above. In the following section, the properties that the new learning mechanism has to fulfill in order to meet these requirements, are discused.

2.4 Features of a desirable learning algorithm

In this section, the features that should be fulfilled by a learning mechanism in order to be applicable in the `hierarchical development` approach and in order to learn general behaviors with deictic sensors and limited perception, are discused.

2.4.1 Learning general behaviors

This property is directly required in our learning mechanism not only because it is the last feature of complex tasks we mentioned above, but also because it is necessary to allow learning of complex behaviors in a hierarchical way.

Let us assume that the agent must learn to solve a given complex task. Following the hierarchical development approach, we have proposed the agent to previously learn other behaviors that later will be used as actions to solve the original complex task. Assume that the complex task is defined in a robotic environment, and that to be solved it is necessary to go out of the room where the agent initially is. In this case, it is proposed, for solving the complex task, to previously learn the *going out of a room* behavior. It seems clear *that if the behavior to be learned for the original task have to be general, then the previously learned behaviors used to learn the complex task, also have to be general.* In our example, if the original complex task is widely applicable, the intermediate behavior of *going out of a room* has to be useful for going out of any room, that is, it has also to be general.

There are two ways of learning the task: by learning a general and unique behavior (for instance, one for going out of any room), or by learning a lot of different behaviors, each one applicable in a different setting (for instance, each one useful for going out of a different room). The learning of a unique and general behavior means the learning of a reusable behavior with the advantages that it implies. Learning a lot of different behaviors involves some inconveniences.

First of all, when the agent learns several different behaviors, the accumulated number of learning experiences performed for learning all of them usually is significantly larger than the amount of experiences needed to learn a single general behavior. Thus, learning several different behaviors usually implies an unreasonable large training.

Secondly, as each learned behavior is added to the agent's repertory of actions, when several different behaviors for solving one task are learned, the resulting number of actions available to the agent is significantly increased. As it is known, this becomes an inconvenience for reinforcement learning because it is preferable to use few actions to learn to solve a task instead of using a lot of them. The more actions that can be chosen in a perceptual state, the more difficult is to choose the right one to execute. One way to solve this proliferation of actions could consist in grouping the actions that correspond to the same general behavior into one single action. When this action is chosen, a procedure (that also has to be learned!) must determine which one among the specific behaviors is applicable in the current state. Unfortunately, the number of steps wasted in the procedure to determine the particular behavior to execute plus the number of steps used in the execution of the particular behavior usually exceed the number of steps that would be needed by a general behavior.

For all these reasons, learning general behaviors is a feature that the a desirable learning mechanism has to fulfill.

2.4.2 Learning behaviors of acceptable performance

Usually, the concept of general behavior is opposed to the concept of optimum behavior. Although general behaviors are more widely applicable, optimum behaviors are by definition more efficient. This is true, because a general behavior (applied to an specific problem) can take advantage only of the features that are common to all the problems it solves, while an optimum behavior, being more specific, takes advantage of all features of the problem including those (that does not appear in all problems) that can lead to a better performance than the produced with a general behavior.

Nevertheless, as we have seen in the previous section, this advantage is not usually exploited. To solve the general task (for instance, *going out of a room*) using optimal behaviors, the agent must know the particular current problem it faces (the current room and the initial position) to retrieve the optimal behavior adequate for this problem. The procedure to determine the particular problem the agent is facing plus the execution of the particular optimum behavior usually takes more time than applying the general behavior. In this case, the optimal behavior execution becomes more costly than the general behavior execution.

However, it could be reasonable to take then the following point of view: although we have seen that to learn an optimal behavior for each particular setting could be not adequate in order to solve a general task, we can still try to learn the best of all the general possible behaviors. In this case, what is the best general behavior remains to be defined. As we will notice later (section 3.5.1), this is not an obvious issue. For instance, the best behavior in the average number of steps for all problems of the task, when applied in the worst problem could result so costly that it could become an unacceptable solution for that problem. On the other hand, a more costly general behavior in the average number of steps could solve all problems in an acceptable way.

Fortunately, the opposition between general behaviors and optimum behaviors does not imply that general behaviors produce bad solutions. In fact, we can design the algorithm to *learn not any general behavior, but only general behaviors that produce good enough solutions for all the problems that compose the task.* What is a good enough solution has to be defined beforehand by the teacher. *This can be seen as learning behaviors that satisfy the task rather than learning behaviors that produce optimum solutions to the task.* This change of focus is necessary because, as we will see in section 3.5.1 for some general tasks, a single behavior that optimizes all the problems of the task does not exist, and the learning of the best behavior in the average number of steps is not always acceptable.

2.4.3 Learning reliable behaviors

The learning mechanism must be able to learn general behaviors and to deal with incomplete information. Also, in order to build a hierarchy of behaviors, it also must show a good performance in learning *reliable* behaviors from the reinforcement signal.

The reliability of the learned behaviors is a feature usually desired, but becomes indispensable in hierarchical learning. For instance, assume that we have learned a 80% reliable behavior. This rate would be usually considered acceptable for a learned behavior, but not if this behavior has to be used as a primitive action to build another more complex behavior.

The reliability of a behavior that solves the complex task depends on the reliability of the previously learned behaviors that it uses as primitive actions, because to obtain a success in a specific problem of the complex tasks it is necessary the success of the previously learned behaviors used in the process of solving the problem. In probabilistic terms, *the probability of success of the behavior that solves the complex task will not be higher than the probability of the successful execution of all the previously learned behaviors used in the process of solving the complex task*, that can be estimated as the mathematical product of the probability of success (reliability) of the previously learned behaviors used in to solve the complex problem.

For instance, assume that the 80% reliable behavior is used to solve the complex task. Assume also that in order to solve the complex task, it is necessary to execute in average 5 times the 80% reliable behavior. In this case, the reliability of that complex behavior will not be higher that 33%! ($0.8^5 = 0.33$, the probability of using 5 times the 80% reliable behavior *being all of them successful*).

Consequently, if the learning algorithm must allow a hierarchical learning, then it is necessary for it to learn reliable behaviors.

2.4.4 Ability to deal with ambiguous perceptions

As we have seen in section 2.2, the use of deictic sensors for solving the problem of a large number of situations, implies a limited perception that usually reports ambiguous information about the current real state. This problem has been called *perceptual aliasing* [206].

Most reinforcement learning algorithms, for instance `Q-learning` [205], assume an agent having complete information about its environment. When these algorithms have to deal with incomplete or ambiguous information, they show a degenerated performance [206], [172].

Thus, an important feature that our new learning algorithm must fulfill is the ability of learning with a limited perception, and consequently, with incomplete and ambiguous perceptions.

There are different approaches to solve this problem, that will be discused in chapter 4. In the same chapter we develop a mechanism in accordance with our learning mechanism and with embedded agents, in which the agent must learn to search and perceive the necessary information to solve the problem[2] and remember it[3].

2.5 Other issues

Other issues have been mentioned in other approaches as related to complex tasks. Some of these issues are tackled also in this thesis, but the structure of the work does not point explicitly to them. A list of these issues is given with some comments and *pointers* to the locations in this work where they are discused in more detail.

[2] This kind of activity has been called *animate perception* [13].

[3] This approach will be called `Active Memorization` and will be presented in chapter 4.

Exploration

Exploration is a sampling method used in classical reinforcement algorithms. It consists in the periodical use of randomly chosen actions, that is, the use of actions that are not in accordance with the current policy. The use of exploration in these algorithms plays three different roles:

The first one is to explore the state space when no positive reinforcement has been obtained. To quickly obtain the reinforcement state, it is important that the exploratory technique will perform a non-redundant search, that is, a search that does not repeat the exploration of states previously explored without success. We do not need an explicit exploration technique for this role. We will address this issue in sections 3.4.2, and 5.1.3.

The second role is to ensure the finding of an optimal policy (behavior). An explicit exploratory technique allows to discover variations that can improve the current policy. As we have seen, we are not interested in optimal policies (see sections 2.2.4 and 3.5.1). Thus, we do not need neither an exploratory technique in this case.

The third role is to ensure an appropriate behavior if the environment presents changes in its dynamics. When a good policy has been learned for the environment but later it changes, the behavior also has to change to fit to the new conditions. To use an exploration technique over the agent's life, ensures that any relevant change will be taken into account for finding a new optimal behavior. In our approach, as we are not interested in optimal behaviors, the only case where the policy has to change is when, due to a change in the environment, the learned behavior is no more 100% reliable (that is, some problems cannot be solved with the learned policy). In our approach, to face this change an explicit exploration mechanism is not necessary. As it will be noted in the next chapter, the new learning algorithm will learn mainly from failures. When the agent fails to solve a problem, the learning mechanism, always activated, will work until another 100% reliable policy is found.

Generalization

Generalization consist in grouping, somehow, the agent's sensory inputs that could have the same action to execute in an acceptable behavior to solve the proposed task. The generalization procedure has two profitable effects; it reduces the size of the spatial resources used to build the behavior and, more important, it allows to speed the learning by suggesting, for a new perceptual state, the same response that is known for another *similar* state (that usually becomes more appropriate than a random guess).

In this work, we do not address explicitly the generalization issue, but it is implicit in the "large number of situations" and "limited perception" issues (see above). In fact, the use of deictic sensors that report incomplete information about the environment implies a forced generalization process. Several different real states perceived with a limited sensory system could be perceived as the same state. This implies a generalization of situations forced by the agent's perceptual system.

Following this idea, some work has been done. Experiences has been performed forcing a generalization of states by means of a limitation of the agent's perceptual system. This limitation can inhibit the possibility of learning to solve the task, but also can increase the speed of learning. Experiences has been performed in in order to empirically find the optimum balance between the two effects. We have done this study applying random neural networks [189] and genetic algorithms [70] to the framework presented through this work. Some results have been published in [110], but a more deep study has to be done.

Long delayed reinforcement

This problem has been discused above under the title "long length of a trial". We recognize it as one of the hardest problems. Our approach is inspired in the concept of development and it has been suggested in section 2.3. It will be deeply studied and tested in chapter 5. Other approaches to ease this problem are also reviewed and criticized in the same chapter.

Hidden state

The hidden state problem appears when not all information that makes the task Markovian is available to the agent. This problem has been partially discused above under the title "limited perception". We recognize it also as one of the characteristic features of complex tasks. Our approach to this problem is called `Active Memorization` (inspired in the *animate perception* approach), and will be deeply studied and tested in chapter 4. Other approaches to solve this problem are also reviewed and criticized in the same chapter.

Teaching

The approach of teaching as a resource for speeding up the agent's learning process is considered in section 5.1.1. There we put forth our doubts about this approach for embedded agents. In chapter 5, other methods, more suited to embedded agents for obtaining this speed up, are presented.

Using external knowledge

The approach of using *a priori* knowledge about the environment as a resource to speed up the agent's learning process is considered in section 5.1.2. There, our doubts about using this approach for embedded agents (which are very close to the presented in the "teaching" topic) are presented. In chapter 5, other methods more suited to embedded agents for obtaining this speed up are proposed.

Incremental learning

Incremental learning, as the ability to gradually learning how to solve increasingly more complex problems from previously learned but simpler similar problems, will be discussed in chapter 5. There, some experiences with our learning algorithm are described.

Hierarchical learning

As we have seen in section 2.4, where the desired features for a learning algorithm to solve complex tasks have been described, we think that the use of previously learned behaviors in the developmental process of the agent, is essential. In chapter 5 our approach to this issue will be explained.

Noise

Noise is a feature that usually arise in real applications. It can appear in the agent's perceptions or in the execution of the actions. We do not explicitly face noise in this work, but experiences described in chapter 6, where a robotic application is presented, show that our approach can work in noisy domains. In the same chapter also can be found some comments about this issue.

Robotic applications

A lot of emphasis has been given to the application of reinforcement learning mechanisms to real applications with robots. Real applications exhibit several features that usually are not present in other applications: *noise*, *efficiency constraints* and *a continuous environment*. Noise has been just considered. Efficiency is a constraint that has to be fulfilled in real applications with robots. It is not realistic to allow the learning process to perform on the order of thousands of experiences, due to time limited constraints. Finally, a robotic environment is usually continuous in two dimensions: time and space. The concept of continuous environment is opposed to the concept of discontinuous environment (represented by a grid).

Experiments performed in a robotic task (described in chapter 6) show that these features do not prevent the successful use of our new approach.

2.6 Summary

In this chapter we have characterized our view of what a complex task is. We have presented a list of the main properties that, we think, are the source of the complexity of a task.

We also have outlined the approach that we will follow to solve the long delayed reinforcement problem, one of the main difficulties for learning complex tasks. This approach, named `hierarchical development` and thoroughly described in chapter 5, is based in incrementally learning, from previously learned behaviors, more complex ones.

Finally, the main requirements for an adequate learning algorithm in our approach have also been described: allowing for a hierarchical learning, learning general and reliable behaviors, and the ability to deal with ambiguous perceptions. Learning mechanisms that fulfill these conditions are presented in the following two chapters.

Chapter 3

The Learning Algorithm: CANDID

> "So saying, they handcuffed him, and carried him away to the regiment. There he was made to wheel about to the right, to the left, to draw his rammer, to return his rammer, to present, to fire, to march, and they gave him thirty blows with a cane; the next day he performed his exercise a little better, and they gave him but twenty; the day following he came off with ten, and was looked upon as a young fellow of surprising genius by all his comrades."
>
> *Candid* or the optimism, Voltaire.

3.1 A new learning mechanism

In this chapter we present a new learning algorithm, called CANDID, developed according to the requirements described in the previous chapter. It must be noted that this learning algorithm fulfills the requirements of the previous chapter, but more complex algorithms following the same direction can be designed.

The learning mechanism we propose, tries to solve a problem by choosing and executing a sequence of actions. At the beginning, the agent will not have any knowledge of what actions to execute in a given situation and, therefore, it will choose randomly actions for execution. New knowledge is acquired by means of a reinforcement signal that indicates whether the proposed sequence of actions solves the problem or not.

Two tables (S_A and F_A) are used by the learning algorithm to store some statistical information necessary for learning the desired behavior. For each possible state of the sensors (situation) and each action, an entry in the two tables is defined with statistical information that will be useful to estimate the *possibility*[1]

[1] We will use informally the word *possibility* only to indicate the chances of occurrence of an

to obtain, in the situation represented by the entry, a positive reinforcement if that action is executed.

Specifically, the statistical information stored in tables S_A and F_A given one state s and one action a, is defined as follows:

$$S\_A(s,a) = \sum_{t=1}^{n} Success(t,s)\ Executed(t,a,s) \tag{3.1}$$

$$F\_A(s,a) = \sum_{t=1}^{n} (1 - Success(t,s))\ Executed(t,a,s) \tag{3.2}$$

where n is the number of times that the agent has faced with the s perception, and $Success(t,s)$ and $Executed(t,a,s)$ are the following functions:

$$Success(t,s) = \begin{cases} 1 & \text{if the } t\text{-}th \text{ time that the agent was faced with the } s \text{ perception, it received a success signal at the conclusion of the sequence execution.} \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

$$Executed(t,a,s) = \begin{cases} 1 & \text{if the } t\text{-}th \text{ time that the agent was faced with the } s \text{ perception, the } a \text{ action was executed} \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

In words: $S\_A(s,a)$ is the number of times that action a has been executed in situation s and a success signal was returned at the end of the trial, while $F\_A(s,a)$ is the number of times that action a has been executed in situation s and a failure signal was returned at the end of the sequence execution. When one action a is executed in one perception s twice in a single execution sequence (because the perception is repeated), the statistics will count twice the fact that action a has been executed for perception s.

The statistical information S_A and F_A will be used to choose the best action to execute in a given situation: *the agent will select, for each perception, the action that presents the higher "possibility" of leading to a success.* We define the *policy* as the behavior described by choosing, for each perception, the action to execute according the above criterion.

event, but without making reference to its strict sense in statistics or possibility theories.

We estimate the *possibility* to get a positive reinforcement when executing an action a, given the actual situation s, as follows:

$$P(s,a) \quad = \quad \frac{S\_A(s,a)}{S\_A(s,a) + F\_A(s,a)} \tag{3.5}$$

This estimation of the possibility of success $P(s,a)$ is meaningful if the future success signal depends only on the execution or not execution of action a. Unfortunately, with delayed reinforcement, success or failure does not depend only on the action executed for perception s but also on the rest of actions executed in the trial. In this case, *the estimation will be meaningful only if the entire sequence of actions executed to solve the problem is obtained from a constant policy* (that means, if for all other possible perceptions, the chosen action is always the same).

Obviously, learning implies changes in the policy (behavior). If we assume that at each trial the agent changes the policy (learns), then we must conclude that the estimation $P(s,a)$ carries equally influential information about the successes and failures obtained using different policies. This would lead to a meaningless estimation of $P(s,a)$, because *the estimation ought to be only about the current policy*. In order to avoid the influence of old policies over the estimation, the number of successes and failures when an action has or has not been activated will be weighted, implementing in this way a kind of "forgetting mechanism". The weighting procedure we use is the following:

$$S\_A(s,a) \quad = \quad \sum_{t=1}^{n} \delta^{(n-t+1)} \, Success(n-t,s) \, Executed(n-t,a,s) \tag{3.6}$$

$$F\_A(s,a) \quad = \quad \sum_{t=1}^{n} \delta^{(n-t+1)} \, (1 - Success(n-t,s)) \, Executed(n-t,a,s) \tag{3.7}$$

where δ $(0 < \delta < 1)$ is a weighting value used to decrease in a exponential way the importance of past experiences, n is the number of times in which situation s has been found, and $Success(t,s)$ and $Executed(t,a,s)$ are the functions described in formulas 3.3 and 3.4.

This exponential decay has been chosen for two reasons. The first one is that it shows a smooth weighting in the influence of past experiences, easily controllable by means of the δ value (that determines the speed in forgetting past policies). The second reason is the easy and efficient implementation of the forgetting mechanism. The reader will note that this definition implies that the weighting procedure is performed for each action (including the non executed actions) each time that perception s is found. The reason will be clear later (see the dynamic α in section 3.5.1).

Initially, without any *a priori* knowledge, we initialize all the statistics with the same value.

$$\forall s, \forall a \qquad F\_A(s,a) = S\_A(s,a) \;:=\; \frac{N}{4} \tag{3.8}$$

where N is a parameter of the learning algorithm. In practice, when the agent faces a new situation, for each action and for this situation, a new entry initialized to $N/4$ is created in tables S_A and F_A.

With the initialization procedure, the definition of the statistics is modified, adding a $\frac{N}{4}\delta^n$ term (that is, the initial value weighted n times):

$$\begin{aligned} S\_A(s,a) &= \frac{N}{4}\delta^n + \sum_{t=1}^{n} \Big(\delta^{(n-t+1)} \; Success(n-t,s) \quad Executed(n-t,a,s) \Big) \\ F\_A(s,a) &= \frac{N}{4}\delta^n + \sum_{t=1}^{n} \Big(\delta^{(n-t+1)} \; (1 - Success(n-t,s)) \; Executed(n-t,a,s) \Big) \end{aligned}$$

If we want to maintain the statistics within a reasonable range of values, we can define δ as:

$$\delta = \frac{N}{N+1} \tag{3.9}$$

This definition of the δ value together with the initialization procedure, ensures that, using the weighting procedure, the statistics will remain in the $(0..N)$ range[2], and that the following sum remains constant over time:

$$\forall s, \forall a \qquad N = F\_A(s,a) + S\_A(s,a) + NoAct(s,a) \tag{3.10}$$

where $NoAct(s,a)$ is a value that represents the no activation index of the action a when the perception s is found. The $NoAct(s,a)$ value is in the range $(0..N)$. The greater the index value, the less used the a action in the s perception. The proof of these statements can be found in appendix A.

Taking all this considerations into account, the learning algorithm *for one trial* that we propose rests as follows:

[2] 0 and N values excluded.

```
chain_solution:= null;
while (not failure) or (not success) do {* problem solving loop *}
begin
  s:= perceive;       {* sets to s the current perceptual state *}
  a:= choose_action(s);
              {* choose the action with the highest possibility
                 to obtain a positive reinforcement from the
                 actual situation. In case there is more than one,
                 choose one of them randomly. *}
  execute_action(a);              {* execute the chosen action  *}
  s´:= perceive;
  while s=s´ do  {* execute the action until perception changes *}
  begin
    s:= s´;
    execute_action(a);
    s´:= perceive
  end;
  chain_solution:= add(<s,a>, chain_solution)
end;                         {* end of the problem solving loop *}

success:= get_reinforcement;
for each <s,a> of the chain_solution do       {* learning loop *}
begin
  for each action a´ do
  begin
    if a=a´ and success      then  S_A(s,a´)= S_A(s,a´)+1;
    if a=a´ and not success  then  F_A(s,a´)= F_A(s,a´)+1;
    S_A(s,a´):= S_A(s,a´) * delta;
    F_A(s,a´):= F_A(s,a´) * delta;
  end
end                                {* end of the learning loop *}
```

In our approach, a `failure` occurs when the agent executes a dangerous action or when certain number of actions (defined beforehand as a parameter of the task) have been executed without success. In both cases, the trial stops with a failure signal. The limit of actions allowed to solve the problem is set by the teacher and it is used to ensure the learning of a good enough (acceptable) behavior (see the requirement of section 2.4.2).

The algorithm described above shows the procedure to follow for learning from a single experience. First, the agent must be set in the initial state. Then, the procedure generates the execution of a chain of actions and, at the end of the execution, the procedure learns from this experience depending on the reinforcement obtained.

To learn a general behavior (see the requirement of section 2.4.1), problems that belong to the general task, are randomly selected and presented as problems to be solved by the agent. These problems are presented to the agent until it learns the desired general behavior.

The learning cycle stops when the agent does not learn anything after a predetermined number of problems is presented. In order to determine when the agent has finished the learning, the current policy is kept in a table. This table keeps the information of the best action to execute for each situation. When the table does not change after a large number of experiences, it means that the agent is not learning anything new and then the learning process must stop. In this case, the knowledge of *how* to solve the task is kept in the policy table which will be used to increase the agent's set of skills. This will allow to learn more complex tasks (see section 5.3).

To ensure the learning of reliable behaviors (see the requirement of section 2.4.3) it is necessary to make a repetition of some experiences. The details and reasons for this mechanism are described in the passage of section 3.2.2 titled "ensuring reliability".

3.2 An example application

A set of experiments have been designed in order to test the performance of the proposed learning mechanism. In this chapter we will show the performance of the CANDID algorithm in one task set in the "blocks world", but the performance of the algorithm in other tasks and other environments can be seen in chapters 4, 5 and 6.

3.2.1 Environment: blocks world

The first environment selected is the "blocks world", that fulfills the required conditions of complexity: incrementally complex tasks can be proposed, the perceptual system gives incomplete or ambiguous reports, the initial set of actions is too simple to solve complex tasks, etc.

This environment was previously used by Chapman [34] and Whitehead and Ballard [206] to show the possibilities and drawbacks of reactive systems in complex domains. Chapman demonstrated by solving the *fruitcake problem*[3] that

[3] How to build, from a set of blocks labeled with characters, a stack where the word "fruitcake" can be read.

reactive systems can accomplish complex goals in this domain without the combinatorial explosion of stimulus-action rules predicted by Ginsberg [69]. This problem is solved using a "deictic" perceptual system that reduces enormously the number of states of the task. Starting from this work, Whitehead and Ballard developed a mechanism to enable *learning* in reactive systems with deictic perception. Chapman has shown that this kind of systems can solve complex tasks, but only if they can operate with an incomplete description of the world. The mechanism Whitehead and Ballard developed, tries to overcome the incapacity of the `Q-learning` [205] method to learn when it is fed with ambiguous information (see section 4.2.2 for an evaluation).

However, the environments described by Chapman [34] and Whitehead and Ballard [206] present several differences with the environment we define. They define the agent with powerful complex actions (for instance, visual routines [202]) for the motor system. These actions are programmed as routines and are given to the agent from the beginning. We do not use so complex actions. In our case, the agent has been designed with a simple set of sensors that reports incomplete information of the world, and a very simple set of actions.

The set of sensors is composed of:

Sensors:

- *Foveal vision:* A sensor that represents the visual focus of the agent and gives information about the content of the visualized zone (ground, block (and its color) or void). This sensor gives only a local information about the environment, limited to the location where the fovea lies.

- *Peripheral vision:* Four sensors that give partial information about the zones surrounding the visual focus. Each sensor indicates only the presence or absence of an object different from void in the zone to the right of, to the left of, above or below the visual focus respectively.

Initially, the set of actions of the agent is composed of:

Actions:

- *Move the visual focus*: Four actions that move respectively the visual focus up, down, right or left one position in the world.

These sets of actions and perceptions in the environment define a non-Markovian environment, due to the local and limited perception of the environment reported by the sensory system.

A standard setting of the environment (see figure 3.1) is composed by a window where there is a table and blocks of different colors resting on it or forming stacks. In the picture we can also note the representation of the visual focus of the agent

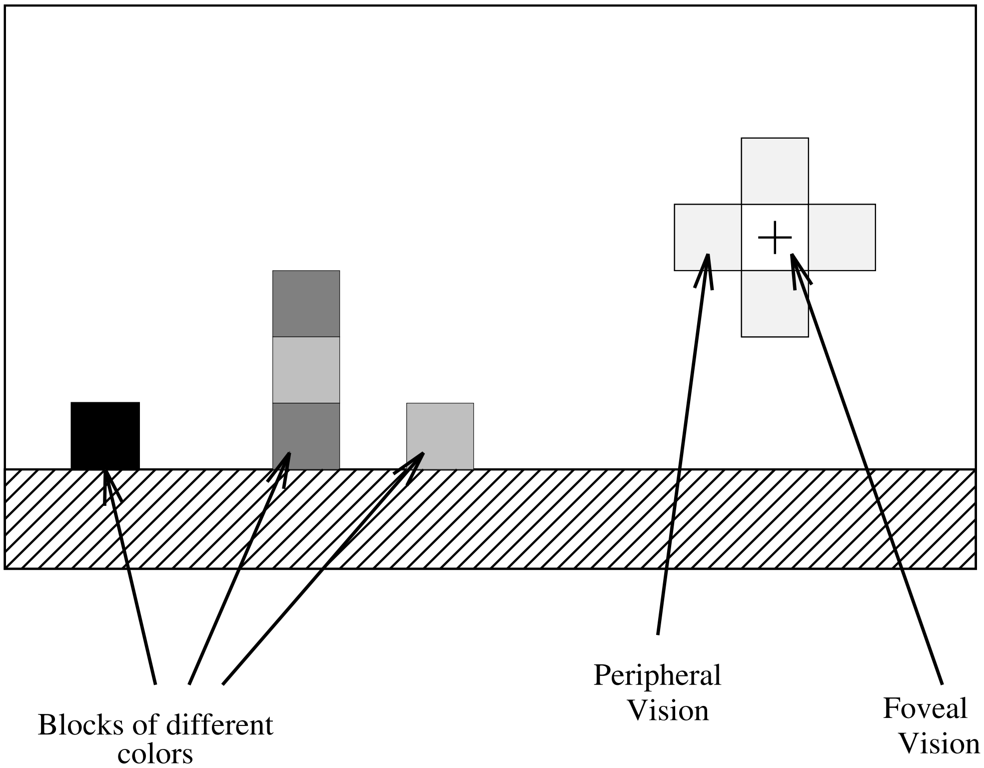


Figure 3.1: A standard setting of the environment.

and its peripheral vision. For the sake of simplicity, the environment has been represented as a grid that allows a discrete and coordinated movement of the eye. The size of the environment is 12x10 (the two bottom rows are occupied by the ground).

The fact that the environment is represented by a grid does not imply that the algorithm needs this kind of representation in order to properly work. In chapter 6 we will see an application of CANDID in a continuous environments. Moreover, if necessary, the current environment could be transformed into a continuous representation.

3.2.2 The task of searching for a block

The first task selected to show the performance of the algorithm consists in searching for a block of a desired color among the set of blocks present in the world.

The behavior learned to solve this task must be general, that is, applicable to different settings (or problems) of the environment. To allow learning of a

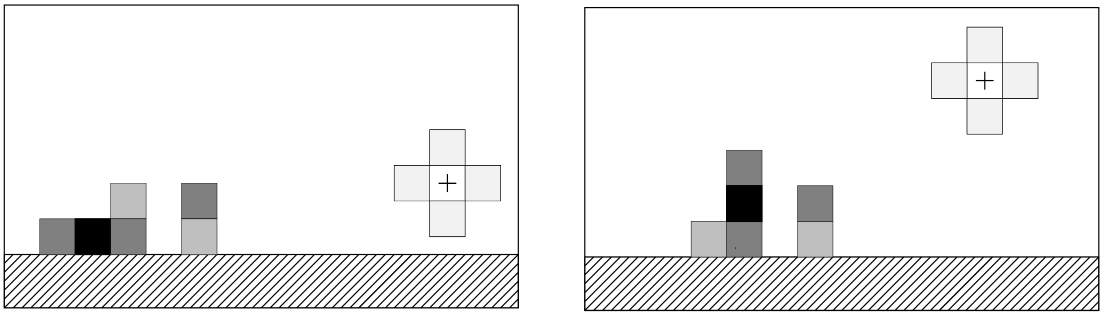


Figure 3.2: Different initial conditions for the general task of *searching for a block*. In these settings, the goal block is the black one.

general behavior, the training of the agent has to be done with different settings. These settings will be randomly generated and will be presented to the agent as problems to solve.

Each randomly generated setting presents a random number of blocks ranging from 5 to 8. The blocks have a randomly chosen color from a set of 4 different colors. To make the problem more difficult, there is only one block of the color we are searching for. In order to learn a behavior useful to find blocks of any color, the color of the block we are searching, randomly changes at each experience.

In order to obtain a general behavior, the procedure that generates the settings of the blocks (that is, that generates the problems of the task) must be able to generate all possible configurations of blocks in the environment (that is, all problems that compose this task). In addition, these configurations have to be generated in a balanced way: qualitative different problems have to be generated with a similar frequency. The procedure that generates the problems of the task achieves this feature by setting each block randomly in the environment with a 50% probability of being set on the ground and a 50% probability of being staked over another block. These probabilities produce an adequate balance between the structures built with blocks that appear in the generated problems; other probability values tend to generate too much or too few stacks or blocks. The only properties common to all settings is that the visual focus begins in a random position in the right zone of the table, while the blocks are situated in the left zone of the table. Two different settings of the task, generated with the above described procedure, can be seen in figure 3.2.

The *searching for a block* task is not trivial, due to the limited perceptual system of the agent and the simple actions it has. To solve this task within a limited number of actions, the agent must take advantage of the invariant features it finds while experimenting in the environment, as for example, the empirical fact that blocks rest on the table or on other blocks. The maximum number of actions allowed to be executed in a single trial is about 25[4].

Although our algorithm is more robust to the perceptual aliasing than other algorithms (see the passage of section 3.5.1 titled "update based on final reinforcement" for a justification) it does not solve the problem. To face this problem we also add, as a perceptual information, a memory of the last executed action. This information will be enough to allow learning with CANDID. *Nevertheless, a more adequate approach to solve the perceptual aliasing problem will be presented in the next chapter.*

3.2.3 Performance

To perform the experiences, we define the following reinforcement function: the agent will receive a positive reinforcement when it perceives with its foveal vision a block of the desired color, but it will receive a negative reinforcement when it does not reach the goal in 25 actions or when the visual focus goes out of the window.

The maximum number of actions allowed to perform an experience has been set intuitively by the teacher. A systematic search for the best value of this parameters has not been performed. The learning cycle consists in presenting randomly generated new situations (settings) of the environment to the agent from which to solve the task.

The first results obtained with CANDID in the *searching for a block* task can be seen in figure 3.3. It represents the number of successes obtained (y axis) versus the number of experiences done by the agent (x axis).

The figure has been obtained in the following way: after each 100 experiences presented to the agent for learning, a presentation of another set of 100 experiences is performed for testing the current policy (avoiding in the test, the learning loop of the algorithm). The average for 30 different runs of the values obtained in the testing phase (after each group of 100 experiences for learning), is shown in the figure.

The difficulty of solving the task proposed also can be noticed in figure 3.3 where the average performance of 10 random behaviors is also shown. The performance achieved by random behaviors (within the limited number of actions) is about 4% of success.

As it can be observed, in this first experiment, the goal of obtaining a reliable behavior is not accomplished; the learning rate is stabilized at the 90% of successes when near of 2.000 experiences has been performed.

In this case, either there is no better policy available for this task (given the limited perception and number of actions available), or the algorithm has not

[4]The number of steps the agent has to perform in order to solve the most complex problems of the task is about 35. Notice that in CANDID, one action is executed until the perceptual state changes. Thus, the execution of one action could involve several steps.

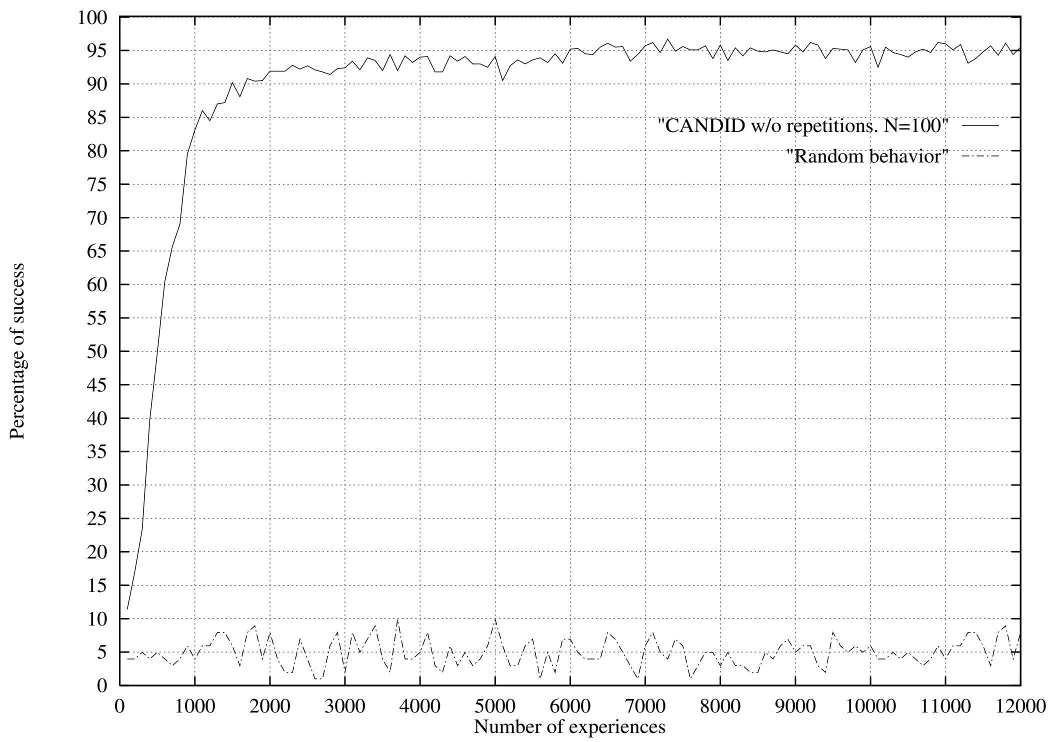


Figure 3.3: Initial performance of the new learning algorithm.

been able to achieve a totally reliable behavior. As we will see in the next section, better policies do exist. Then, CANDID has failed to learn a reliable behavior for the task.

It seems that there are 5% of very complex problems the agent is not able to solve with the learned behavior. The agent has found a good policy and the learning algorithm considers that it is not useful to change the policy for solving this little percentage. Or in other words, the punishment for the 5% of not solved problems is not enough to change a policy that obtains a 95% of successes. In this case, we could say that the learning has reached a local maximum. Obviously, this problem has to be solved.

Ensuring reliability

In a second experiment, in order to achieve a reliable behavior, we apply the following strategy: *if the agent does not solve a problem, then the same problem is presented again to the agent until it obtains a success or the number of repetitions exceeds a predetermined number.* The repetition of the not solved problems forces

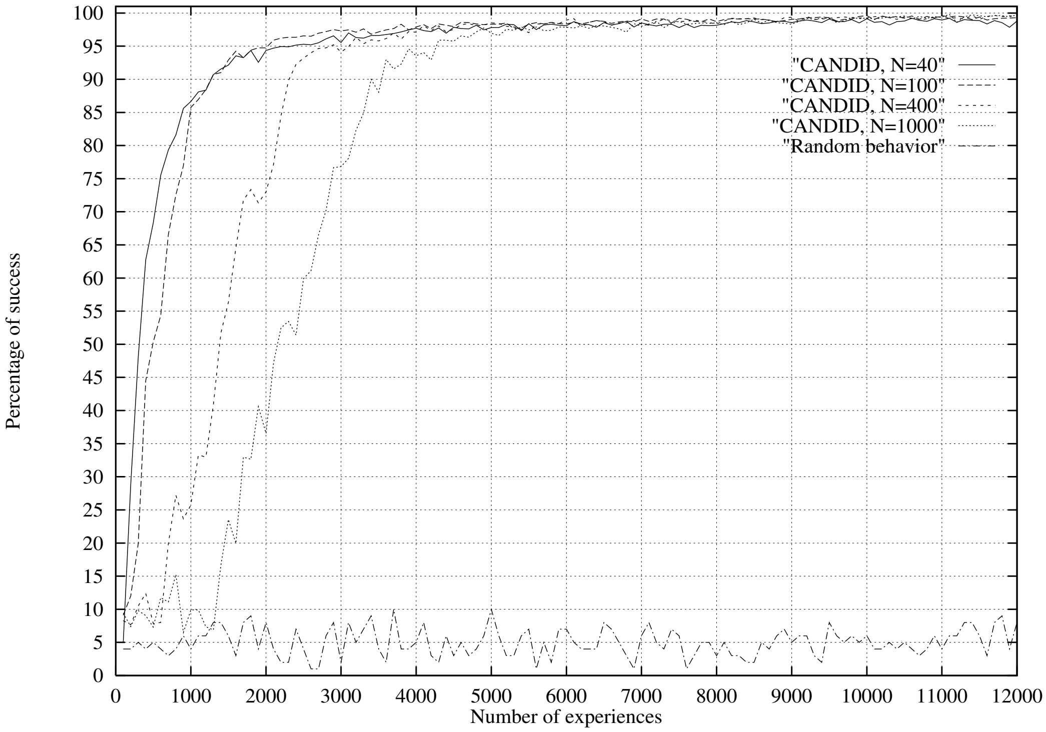


Figure 3.4: Performance of CANDID for different values of the parameter N.

the agent to find a solution for them; with the repetitions, the unusual complex problems appear more frequently and become indispensable to learn.

The maximum number of repetitions is set in order to avoid an initial waste in early experiences and to avoid an over-specialization in the repeated problems. In our experiences we set this maximum number of repetitions to 50 (see the description of the "maximum repetitions" parameter in the following section).

The new results obtained, can be seen in the figure 3.4. Each curve shows the average performance for 35 different runs of CANDID with a different value for the parameter N. There, we can observe the performance for the values of N 40, 100, 400 and 1000. It is varied in order to show the influence of N in the learning performance. The other parameters remain the same as in the previous experiences. It can be seen that the N parameter influences mainly the speed of learning. The lower is the value of the parameter N, the higher is the speed in learning. Another minor influence is observed at the end of the learning. For the lowest values of N, it can be observed that the final convergence to a 100% of problems solved is more difficult. The best performance is obtained when the parameter N is set to 100.

All behaviors learned with CANDID are 100% reliable. CANDID stops the learning process when after 1000 consecutive experiences the agent did not change the policy (see the "maximum number of experiences without a change in the policy" passage in section 3.3 for a discussion about how to choose the adequate value). In this case, we can state that a very reliable policy has been learned. In the proposed task, the learning algorithm always finishes the learning, the most expensive run performing 46400 experiences (although, from 12200 experiences on, the reliability was always above the 96%) and the less costly run performing 2300 experiences.

This could be considered a bad performance because it could seem that, in terms of experiences needed to learn the task, the algorithm becomes very expensive. In spite of this, it must be observed that the learning at the beginning is very quick: from 1000 experiences on, the agent solves more than 86% of the problems and, from 2000 experiences on, it solves more than 95% of the problems.

It must be noted that the learning process at this point could be accelerated if the sequence of experiences proposed to the agent were not selected randomly but following a more smart strategy. For instance, in the 4000 experiences, the agent knows how to solve about 98% of problems. In this case, the 98% of problems presented to the agent for learning does not provide any new knowledge to the agent, because it already knows how to solve them. The only useful experiences for learning are a 2% of the proposed experiences. This explains in part the shape of the learning curve shown in the figure.

In figure 3.5, a trace of the most common general behavior obtained is shown for two different problems. The line shows the trajectory of the visual focus through the environment to find the desired black block. Usually, the learned behavior (shallowly described) consists in: following the table until a stack of blocks is found; to explore the stack; to return to the table, and repeat the procedure until the desired block is found. Some variations of the behavior are also learned to face unusual configurations of blocks (for instance, touching stacks of blocks). Movies showing some behaviors learned for the task are available in the Web at the `http://www-lsi.upc.es/~mmartin/tesis.html` address.

3.3 Parameters for applying CANDID to a task

In this section, a description of all the parameters needed to run the CANDID learning algorithm is made. For each parameter, its role in the learning algorithm is defined. Also, some indications about how to choose the adequate values for the parameters are given.

- **N:** This parameter determines the initial value of the statistics S_A and F_A, and the weighting parameter δ. Usual values of N are in the range [50..200].

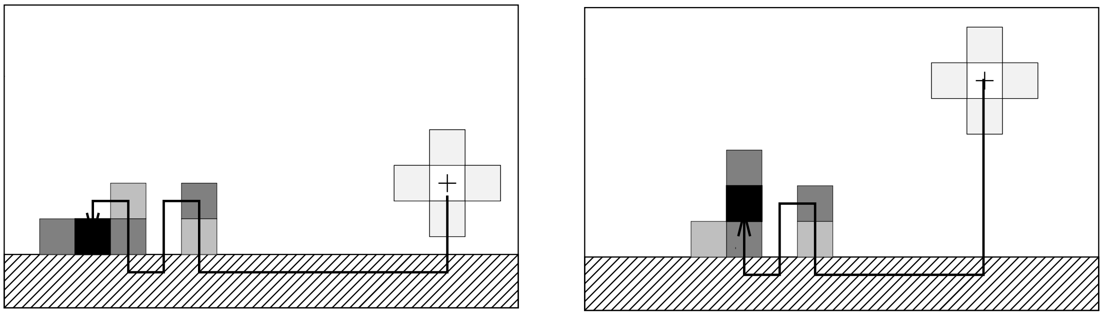


Figure 3.5: Application of the general learned behavior for two different settings. In these settings, the goal block was the black one.

This initial value for entries in F_A and S_A does not represent a bias, as initial `Q` values represent a bias for `Q-learning`. The values used in Candid to select the action to be executed are the $P(s, a)$ of the current perception s, and these values are initially set to $1/2$, independently of value chosen for N.

On the contrary, this value affects the initial value of the parameter that controls the speed in learning (that as we will see later it can be loosely compared to the α parameter used in `Q-learning`). The lower the value selected for the parameter, the higher the initial learning performance. This can be observed in the performance obtained in the experiments applying Candid to the task described in the previous section.

Finally, the chosen value could alter the speed in learning, but a bad choosing usually does not prevent the learning (this never happened with values in the proposed range).

- **Maximum number of actions:** This parameter is used by the algorithm to stop (returning a failure) one experience that with the current behavior needs to execute more than this number of actions to be solved. The failure implies a punishment to the actions taken and a change in the behavior. This parameter is used to ensure the quality of the learned behavior. Obviously, we could define this limit not for the actions chosen but for the number of steps performed in the arena. The choice depends on whether the definition of good enough behaviors is defined in terms of the number of actions or the number of steps. However, those two values are usually very similar.

  The value chosen for the maximum number of actions is not critical. This means that, although we choose a large value, it does not imply that the worst problem, with the learned behavior, needs this number of actions to be solved. In fact this is not what usually happens, as it can be seen in the experiences of the following chapters (see for instance results in table 4.4).

This parameter is not of the learning algorithm, but it depends on the task. Nevertheless, this value has to be provided to the algorithm. The choice of the value has to be done beforehand, and in some tasks (not usually) could be difficult for the teacher to estimate. In this case, a simple and efficient algorithm can be used to find the adequate behavior and the adequate value of the parameter. The proposed algorithm to find this value is an instance of a dichotomic search.

We propose to determine an initial high value k for the parameter and run the learning algorithm. The initial value has to be high enough to allow learning a behavior that solves the task. A too low initial value could prevent learning any behavior. From this initial value, a dichotomic search in the range of $[1..k]$ is performed.

```
k:= initial_value;
i:= 1;
run_candid(k);
while not good enough behavior or k>i do
begin
  mid:= truncate((k + i)/2);
  run_candid(k);
  if behavior is found and it is better than the previous one
      then
         k:= mid
      else
         i:= mid;
end;
return(k)
```

This algorithm returns an adequate value for the parameter (together with the behavior). This algorithm is quite efficient; the number of executions of the CANDID algorithm, in the worst case, is the logarithm of the initial value chosen ($\log_2$ `initial_value`).

It must be noted that this parameter could be more accurately defined. For instance, if the teacher has the adequate knowledge, the value could be determined as the return of a function that depends on the current problem. This allows a closer supervision of the quality of the learned behavior. Nevertheless, using a constant value for the parameter always has given an acceptable performance.

- **Maximum repetitions:** This parameter is responsible for the learning of a reliable policy. As we have seen, the learning algorithm can find a not completely reliable behavior, but so good in the usual presented settings that the punishment obtained for the not solved problems is not enough to change it.

  To avoid this local maximum, we use an strategy that consist in repeating a not solved setting (problem) until it is solved or a maximum number of repetitions of the setting has been performed. This maximum number is precisely the parameter we are defining.

  It is necessary not to repeat a setting an excessive number of times, specially in the first experiences of the agent. An excessive number of repetitions can cause an over-specialization of the behavior on the repeated setting. The over-specialization appears more easily at the beginning of the learning, where few different problems has been tackled. This is the main cause for limiting the number of repetitions.

  These observations could indicate that an adequate approach could consist in making the number of repetitions variable, lower at the beginning of the execution of the algorithm and higher later. This could be an interesting improvement in the algorithm, but requires to know at which rate to increase the number of repetitions. The right value could be determined by considering the percentage of experiences not solved with the current behavior; the higher the number of not solved experiences, the lower the number of repetitions. Nevertheless, it does not seem obvious how to design a procedure able to automatically determine the appropriate value for this parameter. On the other hand, a fixed number of repetitions usually has provided enough good results.

  The parameter is usually set by the teacher to values in the range [50..200], depending on the task the algorithm is applied to. It is easy to detect when the chosen value is too low because, in this case, the learned behavior is not reliable (the number of repetitions is too low to change the behavior when it falls in a local maximum, and then remains not completely reliable). It is also easy to detect when the chosen value is too high because, at the beginning of the learning process, it appears an over-specialization of the first complex problems that makes difficult the subsequent learning.

  Empirically, we have observed that, usually, all values in the proposed range for the maximum number of repetitions generate a reliable behavior. Only in very infrequent cases, values in the range proposed have not produced a reliable behavior. On the other hand, we have observed that the maximum number of repetitions influences the speed of learning, that is, the value chosen can significantly increase or decrease the total number of experiences to be done in order to learn the reliable behavior. Then, it is more important to use the right value for a quick learning of the behavior, than for finding a reliable behavior.

- **Maximum number of experiences without a change in the policy:** This parameter is used by the algorithm to determine when the learning has to be stopped.

  The procedure to stop the agent's learning will be based in the following observation. A good general behavior is not changed, because it obtains a positive reinforcement in all experiences. If we detect that in a predetermined number of experiences, the policy has not changed, then we have an indication that the learned behavior is good. This allows us to implement an automatic procedure to determine when to stop the learning.

  This halting procedure is better than stopping the learning when no failure has occurred in the last n experiences. There are two main reason for this[5]. The first one is that it could happen that the task the agent faces presents noise in the perception of the agent or in the actions executed. Assume that the desired reliable behavior has been found. In this case, a failure in solving a problem could be provoked not by having a bad policy, but by noise. The policy will be punished, but usually it will not change (because we have a good policy and only one punishment is not enough to change it). Then, following our halting procedure, the algorithm could stops the learning process. This would not occur if the agent had used the halting procedure that consists in stopping when the n last experiences are successful. The second reason is that the algorithm also has to detect when does not exist a totally reliable policy in the environment with the proposed constraints. In this case, the algorithm has to stop, preferably returning the better behavior it could find under these conditions. If no reliable behavior exists, then there will be some failed experiences. In this case, if the condition for stopping is that the n last experiences were successful, then the algorithm will not stop never. On the other hand, following our stopping mechanism, the algorithm can stop the learning.

  Nevertheless, this procedure does not totally ensure a 100% reliable behavior. It depends on the value selected for the parameter. The higher the value, the more different settings tested in the last experiences and the higher probability of having a totally reliable behavior when the learning process stops. There are some ways to estimate the right value to set the parameter depending on the amount of error we could tolerate for the task. For instance, let us assume that we set the limit in 200 experiences and that in the last 200 experiences the policy has not changed. Then, we can suppose that in the usual case (being possible to find a totally reliable behavior,

[5]However, in different circumstances than in the ones we will now comment (that is, in the usual case), the two halting procedures usually have an equivalent performance.

and task without noise), the behavior obtained will be above the 99.5% of reliability, because it has not failed not even one experience of these 200. Executing 1000 experiences without a change on the policy indicates, in the usual case, that the behavior is above the 99.9% of reliability, etc.

In addition, the value for the parameter also has to be calculated based on an estimation of the number of qualitative different problems that compose the task. For instance, if the task is composed of only 100 different settings, a good choose for the parameter will be about 150 or 200 experiences, because the reliability of the obtained behavior (which has not changed in 150 or 200 experiences), in the usual case, will be the same that proportion of different problems appeared in the 150 or 200 experiences over all the possible different problems.

- **Variations:** This parameter express if any of the variations of the CANDID algorithm are used. The possible variations are presented in the next section.

It must be noted that the number of parameters needed by CANDID is lower than the needed by other learning algorithms, and that, generally, the values for these parameters are less critical.

3.4 Variations of the CANDID algorithm

In this section, some variations of the basic CANDID algorithm that could accelerate its learning performance, are discused. These variations can be activated in the algorithm through boolean parameters that have to be set to true if the teacher considers adequate to use them for solving a specific task.

For each variation, some comments about when to use it and its effects, are made.

3.4.1 Using variable punishment

We have seen in section 3.2.3 that the repetition of failed experiences is needed to solve problems that, although do not usually appear, must be solved if we want to obtain a general behavior.

One common problem with the repetition strategy consists in that sometimes the punishment for the failure of one experience is not enough to change the policy of the agent. In this case, if the agent repeats the experience, it will return the same sequence of actions that was previously unsuccessful to solve the problem. It happens when the learned policy is strong enough to resist the punishment obtained with the repetition of the same experience.

As we want to obtain a general and reliable behavior, the failed experience has to be repeatedly presented to the agent until the agent is able to solve it. This usually represents a waste in the experiences performed by the agent, because it repeats several times the same sequence of actions until the accumulated punishment is able to change the policy.

A way to efficiently circumvent this problem could be to produce the punishment necessary to change the policy in just a single experience. This implies that we must be able to calculate the number of times t the same sequence of actions will be repeated for the failed experience. Then, we must punish the agent in one single update in a way that the results were equal that punishing the agent in the usual way (adding 1 to the failure statistics) that number of times t.

Fortunately, the number of experiences to repeat until any change in the sequence of actions appears, can be calculated. To obtain this value, while building the sequence of actions for the problem presented, we must keep what we call the *weakest decision*. The *weakest decision* consists in the decision taken for one perception of the sequence to solve the problem in which the difference between the "possibility" $P(s,a)$ of the action taken a (winning action for that perception) and the "possibility" $P(s,a')$ of the second best action a' is the lowest of all the sequence of decisions.

As it is proved in appendix B, the following formula indicates the number of times the same sequence of actions will be repeated if the same failed experience is always repeated.

$$t = \left\lceil \frac{\log\left(\frac{S\_A(s,a) - P(s,a')(S\_A(s,a) + (F\_A(s,a) - N))}{P(s,a')N}\right)}{\log\left(\frac{N+1}{N}\right)} \right\rceil \tag{3.11}$$

where t is the number of experiences for which the agent returns the same sequence of actions, N is the parameter of the learning algorithm, a is the winning action in the *weakest decision* of the chain of actions, a' is the second best action in the *weakest decision* of the chain of actions, and s is the perception of the *weakest decision*.

When this value is calculated, the update of the statistics has to be done as follows in order to obtain the change in the policy *in only one* presentation of the problem.

$$S\_A(s,a) := \delta^t\, S\_A(s,a) \tag{3.12}$$

$$F\_A(s,a) := \delta^t(F\_A(s,a) - N) + N \tag{3.13}$$

This update procedure can be seen as producing a variable punishment (depending on t) that modifies the value of the statistics exactly the same as if the t repetitions of the same problem where presented to the agent (assuming that in the t repetitions the agent repeats the same sequence of actions and fails). The derivation of this formulas can be seen in appendix B.

As we have seen in the previous section, a maximum number of repetitions is set to avoid an over-specialization in the learning process. To avoid this effect when using the current proposed variation of Candid, it is necessary to limit the t value to the same maximum number of repetitions. In this way, we keep the algorithm consistent with the meaning of the parameter. Then we do not take the t value obtained from the above formula, but we set it using the following formula:

$$t \;:=\; max\,(t, MaxRepExps) \tag{3.14}$$

where $MaxRepExps$ is the parameter of Candid previously seen that limits the number of repetitions of one failed experience.

Empirically, experiments show that the number of experiences saved with this approach is not very spectacular (usually about 10%). The t variable usually takes the 1 value and only takes high values (when the savings with the approach would be considerable) in the last experiences (when also the repetition of failed experiences is more useful; see description of the repetition parameter in the previous section).

The use of this variation could become problematic in noisy domains. In noisy domains, a failed experience does not imply a bad policy; the failure could be produced by the existing noise. In this case, to force a change of the policy in only one update, could be erroneous. To adapt this variation to noisy environments, we propose to make decay the t value dividing it by a coefficient β. The t/β value will be the new t value used for the single update. The value must be rounded by excess in order to obtain an adequate integer value for t. The β value will be an integer in the range $[1..MaxRepExps]$, and must be higher as higher the noise is.

3.4.2 Heuristics

In this section some heuristics are presented that also can be useful for speeding up the learning performance of Candid.

H1: Keep same policy in one experience

This heuristic consists in maintaining decisions that have been previously taken over the course of one single experience.

Suppose that we face a problem where, for instance, the third decision has to be done randomly because there exists no difference between the *possibility* of success in the actions for the third perception s. Suppose also that the action a is chosen for this perception. Suppose also that later, for instance, in decision number 15 of the *same experience*, there appears again the same perception s. If we rely on the usual selection procedure, the action to execute must be chosen randomly (because the trial has not been finished and, thus, the $P(s,a)$ statistic for each action and each perception have not changed). *If the action randomly chosen is different from the action chose in the third decision, then for a given perception s there are used two different actions.* This is not allowed in our approach, because the policy (to be consistent) must return always the same action for a given perception.

To solve this inconvenience, in the above described case, we consider that the selection of the action to execute in decision number 15 was made before in decision number 3, where the same perception was obtained. Thus, in decision 15, the same action than the one chosen in decision number 3 must be executed.

This heuristic can be considered as a natural extension of CANDID. In the loop of the algorithm that chooses the actions to execute, choosing a new action when the perception has not changed, is not considered (see above the description of the algorithm). The reason for this strategy is that the selected action for execution has to be the same for the same perception.

It is recommended to always use this variation of the CANDID *algorithm. From now on, in all experiences we will carry out, this heuristic will be used, except when the contrary is explicitly said.*

H2: Only punish/reward one time each perceptual state

This heuristic is used to avoid an excessive punishment/reward of some decisions in the policy being tested. Concretely, the heuristic avoids to punish/reward a decision that involves a perception that was previously punished/rewarded in the same experience.

It usually happens that some perceptions appear more frequently than others. If we punish/reward all the decisions of the sequence without using the heuristic, some decisions (the more frequent perceptual states in the sequence) will be more punished/rewarded than the others. This could distort the learning process. This heuristic avoids this effect by preventing the punishment/reward of perceptual states previously punished/rewarded in the same sequence of decisions.

Obviously, this heuristic can only be used when heuristic H1 is also used, because it guarantees that the action selected for a perception will be always the same in a single experience.

This heuristic is closely related to the *replacing eligibility traces* idea proposed by Singh and Sutton in [174].

H3: Learning from the last random decision

In early experiences, when the agent has not yet acquired so much knowledge, almost all decisions are made randomly. In this case, the agent usually fails to solve the problems and the behavior is punished. This punishment is made to all decisions of the chain of actions proposed to solve the problem, but this is not useful to discriminate between good and bad decisions because all decisions are treated in the same way.

To help in discriminating good from bad actions in the sequence, we will try to punish not all decisions taken, but only some of them. *Heuristic H3 consists in punishing the sequence of decisions only from the last random decision taken.* Random decisions are considered the most probably bad decisions taken (random decisions are usually made when the agent has no experience in the perception, the other kind of decisions is made based on experience). Using this heuristic, punishment is made from the last random decision (that probably is responsible for the failure of the experience) to the end of the sequence of actions executed, but not to the previous decisions (that may be were good decisions).

We must consider the last randomly selected action, according to heuristic H1. For instance, let us assume that we make three random decisions in the generation of the sequence of actions, for instance, in the fifth decision (with perception p_1), in the seventh decision (with perception p_2), and in the tenth decision (again with perception p_1). In this case, the last random decision is not the tenth, but the seventh. We do not consider the tenth decision as the last random decision because the decision for that perception p_1 was made (according to heuristic H1) before (with the fifth decision), and thus, the tenth decision can only follows a decision previously taken.

Another way to discriminate between useful and bad decisions could be punishing only the *weakest* decision made (see above the section about variable punishment). Unfortunately, this is not a good idea, because the weakest decision when punished, becomes weaker. Thus, it will be also the weakest decision in the next experience and also will be the only one punished, and so on. If the weaker decision was not responsible for the failure, the desired behavior will never be learned in this way.

It is important to note that this heuristic is not used when a positive reward is obtained. A success signal implies that *all* decisions taken are the right ones, and thus all of them have to be rewarded. A failed experience implies that *one or more* decisions has been wrongly taken, and then we must punish only them (in the ideal case).

H4: Keep random decisions between experiences if perception has not been punished

This heuristic is an extension of heuristic H1 in order to make the algorithm more systematic in the searching for the desired behavior. In this case, when a random decision is made for an experience, it has to be maintained not only for the same experience but, if the decision has not been punished, also for the next experiences.

Obviously, this heuristic only makes sense when it is used together with heuristic H3, because this is the only case in which one random decision could not be punished after obtaining a failure (it would not be punished if it is not the *last* random decision taken), and then this is the only case in which the decision could be maintained.

H5: Equalize choices when punishment changes the policy

This heuristic is used to make more efficient the execution of the algorithm in case that not any positive reinforcement has been still obtained. When the first positive reinforcement is hard to obtain, most algorithms usually show a bad performance because they redundantly visit states that were previously visited without obtaining success (see section 5.1.3 and exploration in section 2.5). In order to avoid this inefficiency in the CANDID algorithm, heuristic H5 is used.

This heuristic consists in equalizing the possibility values ($P(s,a)$) of the first and second best actions when, due to a punishment, the best action becomes worse than the second best action. When the best action a is tied with other ones and it is punished, if punishment makes the best action worse than the second best action a' excluding the ones tied with the best action, then the possibility value of the punished action a ($P(s,a)$) is equalized to the possibility value of a' ($P(s,a')$).

This heuristic has to be used together with all the previous heuristics to become useful. Its effect consists in not repeating the exploration of any policy, at least until a first positive reinforcement is obtained. A proof for this statement can be found in appendix C.

To make the possibility values equal for the two actions, the update of the action's statistics has to be made in the following way:

$$S\_A(s,a) \;:=\; P(s,a') \quad (S\_A(s,a) + F\_A(s,a)) \tag{3.15}$$

$$F\_A(s,a) \;:=\; (1 - P(s,a'))\ (S\_A(s,a) + F\_A(s,a)) \tag{3.16}$$

These equations make $P(s,a')$ and $P(s,a)$ equal (and then, the two actions a and a' become equivalent for the perception s), but the other properties are kept constant, as the sum $S\_A(s,a) + F\_A(s,a)$ (and thus also $NoAct(s,a)$). This is the desired effect of the actualization.

3.5 Comparison with other works

The proposed algorithm presents several differences with respect to other well known reinforcement learning mechanisms, especially with the `TD` family [180], [205], [17]. In this section, we will make two kinds of comparisons: the first one will be in terms of the theoretic properties of reinforcement algorithms, and the second one will be an empirical comparison for the task proposed as an example in section 3.2.

3.5.1 Theoretic comparison: some observations

In this section, the main features that differentiate the CANDID algorithm from other well known classical reinforcement learning algorithms are shown and discussed.

Non-optimal solutions

Most reinforcement algorithms generate optimum solutions for a task. That is, they learn the best solution for each problem of the task. Unfortunately, this is not always possible when learning *general* behaviors with *incomplete* perceptual information. Under these constraints, the concept of optimization loses its usual sense.

For instance, let us suppose the situation depicted in figure 3.6. As in the task described in section 3.2, the goal is to find a block of a given color. The sensory system reports that there is a block upon the one where the visual focus rests. It also knows that there may be more blocks at the left of the same block, but it does not know where the desired block is. There are two possible actions: `move the visual focus up` or `move the visual focus left`. Which one is the optimal action? Assume that the desired block is on top of the one where the visual focus lies. The best action, then, is to move up the visual focus. It it lies at the left of the block, then the optimal action is to move left. Nor the perceptual information nor the past experiences can help the agent to decide which one is the optimal action, because if the agent is learning a general behavior the two situations can occur.

From this example, we must conclude that under those conditions that a behavior producing optimal solutions for all problems of this task does not exist. If we faced a Markovian task with complete information about the world and a single goal, we could find a behavior that achieves optimally the goal from any initial condition. But without those conditions, this is not possible.

Other works in reinforcement learning, deal with limited perception to allow learning an optimum behavior. Some of them disambiguate the current state where the agent currently is by resorting to a memory of past actions [93], others do it by building a model of the environment [121], [33]. Finally, other works find the best behavior that avoids to visit the ambiguous states [206].

These methods cannot help in the mentioned situation for the task of finding a block. Memory cannot help because we can meet the example situation with independence of the past actions taken. Models of the world cannot help because they are useful only for a specific setting, and solving a general task implies solving

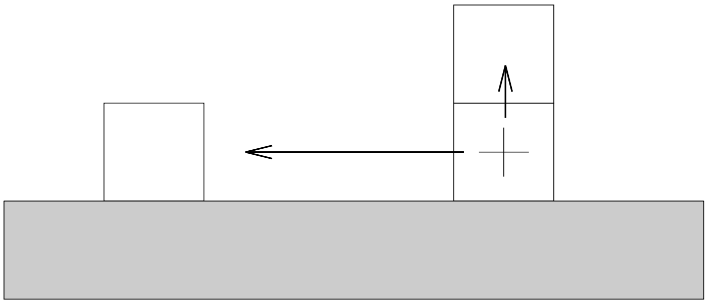


Figure 3.6: There is no optimal action.

it for different settings. Finally, avoiding ambiguous states could be impossible to do in some cases if we want to learn a useful behavior, for instance in the case example. We will see in section 4.2 more detailed study of these approaches.

Another possibility in order to still learn a general optimal behavior could consist in considering the best behavior (the optimal) the one that needs less steps in average of all possible problems to solve the task. This could be intuitively right for Markovian tasks, but it is not adequate in tasks with ambiguous information. In these conditions, the best behavior in average for all problems could be so costly when applied in the worst case, that it would become unacceptable[6].

Thus, we approach to this concept in a different way. We will not search for optimal behaviors but for acceptable behaviors. The agent will learn the behavior under a set of constraints including the reasonable cost to solve the problem. If the problem is not solved within a predefined number of actions, the trial is halted with a failure. This procedure ensures that if the learning algorithm stops returning a behavior, it will be a good behavior (low cost) for all problems of the task (although not the best for them because such behavior does not exist).

We think that the best approach to work with general behaviors and limited perception, is to learn the behaviors that are acceptable for all problems of the task.

Exploration

Another major difference between CANDID and other algorithms, as the ones in the TD family, is that the former *does not need an explicit mechanism for exploration.*

[6]We have empirically found that in some experiments, the best behavior found in terms of the average number of steps needed to solve the task is unacceptable (that is, it exceeds the maximum number of steps allowed) for some problems of the task.

Exploration is used to *optimize* solutions learned by the agent. Although the learned policy guides the agent to the goal, it is necessary to test actions that are not in the policy in order to find, if possible, lower cost policies. In our case, we are not looking for optimal behavior and thus, we do not need exploration.

Exploration is also used to adapt the learned behavior to changes in the dynamics of the environment. If the dynamics change, some modifications of the learning algorithm have to be introduced to find the new optimal behavior for the new conditions. In our case, when the environment changes, we do not need to change the behavior (because we are not looking for optimal policies) except in the case that with the change, the learned behavior does not maintain the reliability. This case is easily detected because failed experiences begin to appear. When the failed experiences begin to appear, the learning mechanism, always activated, will work until another 100% reliable policy is found. Thus, in this case, we do not need a mechanism for exploration either.

Finally, exploration mechanisms are also used to smartly explore the state space when positive reinforcement has never been obtained. To quickly obtain the first positive reinforcement, some smart exploratory techniques are used that perform a not redundant search of the state space (see section 5.1.3). In our case, it can be proved (see appendix C) that using the set of five heuristics discused in section 3.4.2 with CANDID, the execution of the algorithm develops a non redundant search in the space of policies (not repeating the testing of a failed policy) until the first positive reinforcement is obtained. Thus, in this case exploration is not needed either.

Forgetting mechanism as a dynamic α

The forgetting mechanism used by CANDID is not implemented in other learning algorithms, but forgetting exists in a different way. The usual form of the update procedure in `TD` algorithms (for instance `Q-learning`) is the following:

$$Q_t(s,a) = \alpha_{t-1} Z + (1-\alpha_{t-1})\, Q_{t-1}(s,a) \tag{3.17}$$

where Z is a value depending on the current reinforcement obtained and the estimation of future reinforcement possible to obtain from the next states if the policy is followed (see, for instance, [83]), and t is the number of times that perception s has been faced.

What is important in this formula is the α_t value that is used to weight the recent and old past estimations to determine the current estimation. The α_t value has to be in the range [0..1]. The higher the α_t value, the higher the weight of recent experiences.

Although in the general form, the α value depends on t, in practical applications it usually takes a constant value. Also it is known that with a value of

$$\alpha_{t-1} = \frac{1}{t-1} \tag{3.18}$$

all experiences (past and recent) influence equally the current estimation.

In appendix D, our approach is reformulated so as to allow comparison with this forgetting mechanism. Surprisingly enough, we obtain the following equation that express the evolution of $P(s,a)$ over experiences, that is close to TD reinforcement algorithms:

$$P_{t+1}(s,a) = \alpha_t(s,a)K_t + (1-\alpha_t(s,a))\,P_t(s,a) \tag{3.19}$$

where K_t indicates in the current experience whether there has been a success (returning 1) or a failure (returning 0), and the $\alpha_t(s,a)$ value is defined as follows:

$$\alpha_t(s,a) = \frac{A_t(s,a)}{S\_A_t(s,a) + F\_A_t(s,a) + A_t(s,a)} \tag{3.20}$$

where $A_t(s,a)$ indicates whether in the current experience the action a has been chosen for perception s (returning 1) or not (returning 0), and the S_A and F_A are the current statistics for action a and perception s.

As it can be noted, the value of α depends on the sum of $S\_A(s,a)$ and $F\_A(s,a)$ that, as we can deduce from formula 3.10, indicates the index of activation of action a for perception s. Then, our α value changes dynamically depending on the current activation $A_t(s,a)$ and the index of habitual activation $S\_A_t(s,a) + F\_A_t(s,a)$ of the action a for the perception s. The α value can vary in the range $(0..\frac{1}{N+1})$. On one hand, when the action has not been executed, the α value becomes 0 and the $P(s,a)$ does not change (although it changes the index of activations $S\_A_t(s,a) + F\_A_t(s,a)$). On the other hand, when the action is executed, the lower the sum $S\_A_t(s,a) + F\_A_t(s,a)$, the higher the α value, and then, the higher the change in $P(s,a)$.

This implies that if an action with a low index of activation (not recently used) is selected and becomes a success, then its $P(s,a)$ value will be considerably increased. If the action executed is usually chosen (it has a high index of activation) and becomes a success, the increase in the $P(s,a)$ value will not be so high than in the preceding case. In other words, the forgetting mechanism used by CANDID tends to comparatively increase more the $P(s,a)$ value for not recently used actions, than for recently used actions.

The main cause of this difference between the forgetting mechanisms of TD reinforcement learning algorithms and CANDID is that TD *methods only actualize statistics of the activated actions for each situation faced, while* CANDID *updates statistics for the all the actions, activated or not for each situation faced*, as can be seen in the code of the algorithm (the statistics are always weighted).

We think that this different forgetting mechanism is the main responsible for the better learning performance shown by CANDID when it is compared with other known algorithms (we will observe this in section 3.5.2).

Update based on final reinforcement

The CANDID algorithm updates the statistics for each perception and action (that is, learns) using the final reinforcement obtained with the whole chain of actions. This kind of update is used in some other reinforcement algorithms, specially in `TD(1)`, that is, `TD` algorithms with the λ parameter set to 1.

The use of only the final reinforcement to base the learning process has proved to be fruitful in the kind of tasks we are addressing. Specifically, *this kind of update makes the algorithm robust to the perceptual aliasing problem because the algorithm does not base learning on the estimation of reinforcement of the next perception (that could be an aliased state).* In [172], some justifications for this statement are shown. In an empirical comparison of CANDID with other algorithms (see section 3.5.2), we have verified this statement: instances of `TD(0)` (that are not based on the final reinforcement) failed to obtain general and reliable behaviors.

Policy based vs. value based

Our algorithm is not interested in obtaining good estimations of the future reinforcement returns, as in other works, because we are not interested in optimal policies. We are interested in finding good policies as quickly as possible. Thus, the values stored in the S_A and F_A tables, which determine the current policy, are only useful to indicate of the relative importance of taking one action with respect to the others.

In this sense, most of the variations of the basic algorithm shown in the previous section, are oriented to quickly find an adequate behavior by using values *as tools* for finding it. For instance, heuristic H5 manipulates the values of statistics (it equalizes the values for two actions when one becomes better than the other one) in order to make more systematic the search in the space of policies.

3.5.2 Empirical comparison

In this section, an empirical comparison, for the task proposed in section 3.2, between CANDID and other known reinforcement mechanisms, is made. The selected reinforcement learning algorithms for the comparison are `Q-learning`, `backward Q-learning` and an instance of `TD(1)` learning: `Sarsa`(λ) with $\lambda = 1$.

For comparison purposes, the first algorithm we choose is a well known version of `TD(0)`: `Q-learning` [205]. `Q-learning` is designed to find optimal solutions. As we have seen in the previous section, in cases where general tasks must be learned with an incomplete perception of the environment, optimal behaviors do not exist (at least in the way defined for `Q-learning`). To allow a comparison with CANDID, we have set the parameter γ (called also discounting parameter; see definition of optimal behavior in section 1.2) of `Q-learning` to 1[7]. With this value for the parameter, all behaviors that solve the problem, become equally good. We avoid, in this way, the search for an optimum behavior.

Given that the task proposed for the comparison is based on the consecutive execution of trials and given that reinforcement (positive or negative) is obtained within a limited number of actions, we can use for the task the `backward` version of `TD` algorithms. In these algorithms, the updating of statistics is done backwards *after* the finishing of the trial. It is known that the `backward Q-learning` [92] propagates the reinforcement information more quickly than `Q-learning`. Thus we choose also this algorithm for the test. The algorithm, executed off-line, also can be viewed as a version of `TD(1/n)` (that is, `TD` with the λ parameter set to $1/n$, where n is the number of times the current perception has been faced); see [188] for the proof. As in the previous case, we set the γ parameter to 1 in order to prevent the agent to search for an optimal policy that does not exist.

The last reinforcement learning algorithm that we have selected for the comparison is the `Sarsa(1)` algorithm [158], that is, a version of `Q-learning` that includes eligibility traces with the parameter λ set to 1. It learns the values of the current policy in the way that `TD(1)` do it. In the previous section, we observed that the information in which the update of the statistics in `TD(1)` is based, as in the case of the CANDID algorithm, is the final reinforcement signal obtained. Thus, it would be interesting to compare this algorithm with CANDID. As in the previous case, for comparison purposes, we set the γ parameter to 1 in order to prevent the search for an optimal policy that, in fact, does not exist.

In the same way that the γ parameter, in all algorithms we use for the comparison, any exploration technique has been excluded, because, as we have seen in the previous section, exploration is not adequate for learning general and reliable behaviors. We achieve this by setting to one the parameter that determines the probability of taking the policy action at each decision step. Thus, all actions will be taken following the policy[8].

In all algorithms, the initialization of the table of values is done by setting to 0 each entry. Also, a positive reinforcement of 10 units is given to the agent when it achieves the goal, but it obtains a negative reinforcement of -5 units when the agent does not solve the problem in the maximum number of allowed actions. In the present framework, unlike other frameworks, the balance between the amount of positive and negative reinforcement is not critical for obtaining the

[7] The fact that γ has been set to 1 does not prevent the convergence to this task. It has been proved [78], [53] that `Q-learning` with $\gamma = 1$ converges when all policies converge to a zero rewarding absorbing state. In this case this is fulfilled, because all policies are stopped when a predetermined number of action has been executed.

[8] In this case, `Sarsa(1)` becomes equivalent to `Q(1)` [148].

desired behavior. The reasons for this are, in this case, that not any immediate reinforcement is given to the agent and that the γ parameter has been set to one.

One critical parameter for the algorithms chosen to perform the comparison, is the α parameter (usually called *learning rate*). We have decided to use the *search-then-converge* proposal [50] to set it:

$$\alpha_t(s,a) = \frac{\tau \alpha_0}{\tau + n(s,a)}$$

where $n(s,a)$ in the number of times that perception s is faced and action a executed. Notice that the value of $\alpha_t(s,a)$ depends on the parameters τ and α_0, that are the same for each pair perception-action. These two parameters have to be given by the teacher. Notice also that when τ is set to very high values, the value α_t tends to remain constant.

In order to make a complete empirical comparison with CANDID, we have run the `Q-learning`, `backward Q-learning` and `Sarsa(1)` algorithms with different values for the parameters. For each algorithm we have performed 630 runs, that is, ten runs for each combination of the following parameters:

- **Parameters for `Q-learning`:**
  - τ (7 values): 1, 10, 100, 1000, 10000, 100000, 1000000
  - α_0 (9 values): 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
- **Parameters for `backward Q-learning`:**
  - τ (7 values): 1, 10, 100, 1000, 10000, 100000, 1000000
  - α_0 (9 values): 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
- **Parameters for `Sarsa(1)`:**
  - τ (7 values): 1, 10, 100, 1000, 10000, 100000, 1000000
  - α_0 (9 values): 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
- **Parameters for Candid:** We use the same parameters that were used in section 3.2.

An execution of the algorithms without repetition of experiences showed that none of them achieve a totally reliable behavior (this also happened with the CANDID algorithm). In order to overcome this problem, the same strategy that was used with CANDID, is used now: *when a problem is not solved, it is presented again until it is solved or until a predetermined number of failed repetitions is exceeded.*

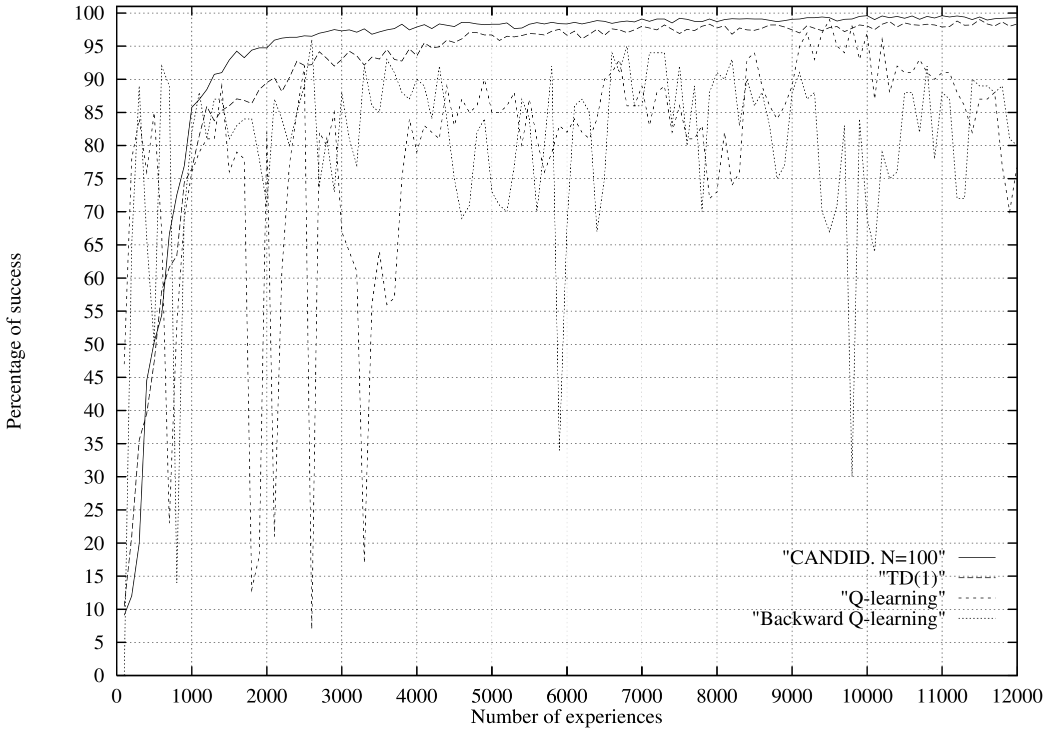


Figure 3.7: Comparison of the CANDID algorithm with `Sarsa(1)`, `backward Q-learning` and `Q-learning`.

A second execution, now with the repetition of failed experiences, has been made. The maximum number of repetitions for a not solved problem has been set to 50. The learning stops when, after 1000 consecutive experiences performed, the agent has not changed its policy. Results for this second execution can be seen in figure 3.7. Each curve shows the average results in 10 runs for the set of parameters that produces a better performance of the algorithm, except the `Q-learning` and `backward Q-learning` curves that, in order to show their particular oscillatory behavior, show their best performance for a single run.

In the figure, we can notice that only CANDID and `Sarsa(1)` algorithms can find reliable behaviors, while `Q-learning` and `backward Q-learning` show a non stable performance.

On the other hand, although both `Sarsa(1)` and CANDID achieve reliable behaviors, it must be noted that CANDID is two times faster than `Sarsa(1)` in learning the behavior. When CANDID achieves a determined percentage of problems solved after a number of experiences performed, `Sarsa(1)` achieves the same level but after performing a double number of experiences.

From this empirical comparison, we can conclude that CANDID is better suited to learn reliable and general behaviors than the other tested `TD` methods. Only `Sarsa(1)` after pertinent modifications (setting γ to 1, repetition of failed experiences, no use of exploration, etc) becomes able to learn reliable behaviors (mainly because the modifications turns the `Sarsa(1)` algorithm closer to the CANDID algorithm). In addition, although `Sarsa(1)` can also achieve reliable behaviors, CANDID has shown a more quickly performance in learning them.

3.6 Conclusions

In this chapter a new learning algorithm, CANDID, that fulfills the properties required by the approach outlined in the previous chapter, has been presented and tested. This new algorithm presents some novel features compared with previous reinforcement learning algorithms.

First, this new algorithm implies a change of focus in the usual reinforcement learning algorithms, because it has been designed to learn general and reliable behaviors. This contrasts with the usual approaches that learns optimal solutions for specific problems.

Learning general behaviors is more difficult than learning specific behaviors, because the agent can use for learning only features that are present in all problems of the task. Sometimes, the algorithm finds a good policy for the most usual problems that is not totally reliable. In this case, it would fail for some rare problems. Nevertheless, the policy found is so good for the other problems, that the algorithm becomes reluctant to change it. To learn also these rare problems, and return good general and reliable behaviors, a new strategy that consists in the repetition of failed experiences, is proposed.

Another feature the new learning algorithm fulfills is that it is robust to the perceptual aliasing problem. CANDID has been designed to allow solving tasks even under a low degree of perceptual aliasing. It has been achieved by basing the learning procedure on the final reinforcement obtained. Although CANDID is not able to completely solve the perceptual aliasing problem (we will see in the next chapter a new technique that we propose to solve this problem), the robustness of the algorithm allows to decrease the amount of resources needed to solve it.

A novel strategy, incorporated in the learning mechanism, allows to automatically determine when no improvement in the learning process is obtained. Then, the algorithm is able to automatically detect when the learning process for the general behavior has to be stopped. We show that this new procedure, based on the stability of the policy obtained, is better than other more naive procedures that determine the end of the learning process based on the number of failures in the last experiences. This feature is required for the new hierarchical learning mechanism proposed in chapter 5.

Another new feature of the learning algorithm is its weighting procedure that actuates, compared with other classical approaches to reinforcement learning, as a variable learning rate parameter α. It produces a considerably acceleration of the learning process. Some adaptations of this procedure could be applied to other classical reinforcement algorithms, for instance in `Q-learning`.

We have shown that all this new features of the learning algorithm become useful for learning general and reliable behaviors. CANDID has been successfully tested in a test task. Other known reinforcement algorithms have failed to solve this task or have solved the task at least two times slowly than CANDID.

Other new contributions consist in variations of the CANDID algorithm proposed to accelerate its learning process. Some of these variations could be successfully applied to classical reinforcement learning algorithms (for instance, to `TD` methods), specially the *variable punishment* and heuristics `H1` (only for trial-based tasks) and `H2`.

Heuristics `H3`, `H4` and `H5` are designed in order to made the learning process performed by CANDID more systematic. When they are used together with heuristics `H1` and `H2`, another interesting feature of CANDID appears: when the whole set of heuristics is used, the algorithm does not redundantly explore policies previously tested unsuccessfully, at least, until a first positive reinforcement is obtained. This statement is proved in appendix C. This implies a high acceleration of the learning process performed by CANDID. Classical reinforcement learning algorithms, as `Q-learning` or `TD` methods, also tend to repeat the exploration of no successful states. Some smart exploration mechanisms have been proposed in order to avoid this. Unfortunately, these exploration mechanisms are not useful for tasks with perceptual aliasing. The set of heuristics we propose, plays for CANDID the same role as these exploration mechanisms for classical reinforcement learning algorithms. The heuristics change the techniques that avoid to visit previously explored states unsuccessfully (techniques that are not useful for tasks with perceptual aliasing because these states cannot be discriminated) by techniques that avoid to test previously tested policies. These heuristics are, then, a new contribution to reinforcement learning because they define the first mechanism (we are aware) that is useful for tasks with perceptual aliasing for avoiding the repeated testing of failed policies.

Finally, it is also interesting to note that CANDID needs less parameters for learning than other reinforcement algorithms, and that the values chosen for these parameters are easy to find and not critical (that is, a bad choice of the values of the parameters, usually does not prevent learning a behavior, but only slightly influences learning speed or the efficiency of the found behavior).

In spite of all these features of the new learning algorithm, other issues of our approach are debatable and must be commented.

In order to find a reliable behavior, we have applied a new strategy based on the repeated presentation of failed experiences until they are solved or until a maximum number of repetitions has been exceeded. Although this strategy has been useful for CANDID, it must be noted that not in all kind of tasks it is realistic to apply it. Concretely, in order to repeat an experience, the initial conditions of the problem must be restored. In some cases it could be difficult to do. In most reinforcement learning algorithms, a similar condition (but no so strong) that requires the agent to be located in a randomly chosen initial state after the execution of a trial, is needed.

Nevertheless, in some cases, the initial conditions could be automatically restored, mainly when each action of the agent can be reversed. In these cases, in order to repeat a failed experience, the agent restores the initial conditions by executing the inverse of the sequence of actions executed by the agent to solve the problem.

Another issue debatable in our approach is that CANDID can only be directly applied to the kind of tasks that can be defined as achieving a goal state. Another kind of tasks are defined as the permanent fulfilling of one condition, for instance the *survive* task is defined as the permanent fulfilling of the *be alive* condition. To face this kind of tasks, the goal condition has to be redefined in terms of the first kind of tasks (that is, in terms of achieving a goal condition).

Another debatable issue is the *maximum number of actions* parameter. In classical reinforcement learning algorithms, this parameter is not needed because the algorithm learns the optimal behavior. It implies that these algorithms also learn which is the adequate number of steps. In CANDID, this parameter has to be given to the agent. Fortunately, that number could be easily estimated by human teachers, or calculated by mechanisms similar to the one described in section 3.3.

Finally, we must note that there are several aspects of our approach that, clearly, must be improved.

One of them is that, although CANDID is robust to the perceptual aliasing problem and even can solve tasks under a low degree of perceptual aliasing, it does not totally solve this problem. In the experiences, we have added to the agent the last action executed as perceptual information. We consider that this is not a satisfactory approach. In the next chapter, a new approach to solve the perceptual aliasing problem, suited to CANDID, is described.

Another problem of CANDID is that it is not able to learn from a very long delayed reinforcement. Delayed reinforcement becomes a problem in classical reinforcement algorithms, but it can be solved applying some mechanisms, for instance smart exploration mechanisms [195]. Unfortunately, exploration mechanisms cannot be applied in tasks where also appear the perceptual aliasing problem. In order to solve this, we propose in chapter 5 a hierarchical approach to solve the long delayed reinforcement with CANDID.

Finally, in this approach, we have not explicitly addressed some features that usually are fulfilled in real applications, for instance in robotic tasks. Two of these features are the presence of noise and the interaction with a spatial and temporal continuous (not gridded) environment. In chapter 6 we show the performance of CANDID in a robotic task that presents these features. It must be remarked that no significant modifications have been introduced in CANDID in order to face these features.

Chapter 4

Active Memorization

"Pensé que cada una de mis palabras (cada uno de mis gestos) perduraria en su implacable memoria; me entorpeció el temor de multiplicar ademanes inútiles."

Funes el Memorioso,
Jorge Luis Borges.

4.1 Introduction

One of the most important problems in reinforcement learning appears when the agent has to deal with ambiguous or incomplete information about the environment (see sections 2.2.2, 2.2.3 and 2.4.4). When the current information provided by the agent's perceptual system is not enough to allow the right choosing of the next action but it could be determined with additional information about the past actions and/or perceptions, then it is said that the task *lacks the Markovian property* in the interface between the agent and the environment (see the definition of Markovian property in section 1.2).

In this case, for one perceptual state, it will be better to execute sometimes one action and sometimes another one. For instance, assume that the information reported by the sensory system, for two different real states[1], due to the limited perception of the environment, becomes the same perception p. Assume also that for this perception p, the action to execute in any policy that solves the task, depends on the *real* state where the agent is. If information about past actions and/or past perceptions could help in order to choose the right action to execute, then the task defined by the set of perceptions and actions of the agent is *non Markovian.*

[1]We denote *real state* as the state described with the complete information (opposed to the perceptual state concept).

In addition, when the current information delivered by the perceptual system is not enough to obtain an unambiguous description of the current situation, it is said that the agent suffers a *perceptual aliasing* of situations [206]. This problem becomes critical when the disambiguation of some states is essential to solve the task.

Finally, if the original task (with all the information about the environment) fulfills the Markovian property, the task (with limited perception) is also called a *partially observable problem* that can be modeled by a *partially observable Markov decision process* (POMDP) [19], or a Markovian problem with *hidden state* [125].

Most reinforcement learning algorithms, in order to work properly, assume the Markovian condition but, unfortunately, this condition is not fulfilled in quite a lot of domains. For instance, it is not realistic to suppose that a robot's sensory system could report all the relevant information about the environment in a single perceptual state. Other common cases where the Markovian property is not meet, are those tasks with agents that have a deictic perceptual system, because this kind of perceptual system usually implies the *perceptual aliasing* problem.

In this chapter we review the approaches to reinforcement learning dealing with the perceptual aliasing problem and propose a new mechanism called `Active Memorization` in order to solve it[2]. The new approach has been tested in several well known domains and results will be shown for each one of these cases.

4.2 Previous Works

There have been different approaches to overcome the perceptual aliasing problem. We classify the different attempts in the following sections.

4.2.1 "Ignoring" the problem

Some researchers apply their reinforcement algorithms in the hope that although there exists a theoretical problem, it will not appear in most domains. In the worst case, if the symptoms do appear, the interface between the agent and the environment could be modified by supplying the agent with additional perceptual information to solve the impasse. This approach works in some cases, but not in all of them. Theoretically, it has been stated [172] that errors produced by perceptual aliasing in `TD(0)` methods can become arbitrarily high. Empirically it has been shown that there are some algorithms (e.g. `TD(1)`, `Monte Carlo` and also CANDID) more *robust* to that problem[3] (see section 3.5.2 and [146]), but

[2]When we say that one approach solves the perceptual aliasing problem, we mean that the approach allows to learn a behavior that solves all problems of the task. This does not mean that the approach is able to disambiguate all ambiguous perceptual states, but only the perceptual states that must be disambiguated to solve the problem.

[3]Because the evaluation of states is guided directly by the final reinforcement signal.

they cannot solve the problem either. Also it is important to note that it is not always feasible to add perceptual information to the agent in order to avoid the perceptual aliasing problem.

Other approaches consist in finding, by policy improvement, *suboptimal* deterministic policies given the perceptual aliased states [79]. Unfortunately, suboptimal solutions are good in some cases but not always. Suboptimal means better than neighbor solutions, but this does not always imply a good enough solution (as we use in the CANDID framework; see sections 2.2.4, 2.4.2 and 3.1).

Other researchers have stated that the *stochastic* property of their algorithms in the action choosing process, can overcome the perceptual aliasing problem. These algorithms learn a probability value of execution for each action that is proportional to the proportion of real states confused in the aliased perception in which the action is the adequate one. For instance, if three different states s_1, s_2, s_3 are aliased into a single perception p, and actions a, b, a are, respectively, the adequate ones for states s_1, s_2, s_3, then the stochastic algorithms will learn a 2/3 probability of execution for the a action (because it is the appropriate action in two of three possible cases) and a 1/3 probability of execution for the b action (because it is the appropriate action in one of three possible cases). As it exists a non-zero probability for the execution of the adequate action, when the perception p is obtained, sooner or later the adequate action will be executed.

This approach is suggested, for instance, by Cliff and Ross in [46]. Its main advantage is that there is no need to disambiguate aliased perceptual states. On the contrary, the behaviors obtained by using this approach are are less efficient than the ones obtained with other approaches. We think that this approach could be useful in some cases but *it does not solve the problem.* This approach is not adequate, for instance, in dangerous environments, where the execution of an inadequate action can become fatal. Another problem of this approach is that as the perceptual aliased states grows, the behavior produced by the stochastic policy becomes more similar to a totally random behavior.

4.2.2 Avoiding ambiguities: Animate Perception

One of the first proposals for solving the perceptual aliasing problem consists in developing policies that avoid to pass through ambiguous states. If ambiguous states are misleading for the agent, then one obvious approach would consist in avoiding these states. This approach implies that, in order to avoid these states, the agent controls the motors of the perceptual system (for instance, the muscles that control the movement of the eyes in humans). When it happens, it is said that the agent has an *animate perception* [14].

This approach tries to develop a policy to solve the problem in the usual way by estimating the future reinforcement for each perceptual state. Sometimes, a perceptual state p that obtains very different reinforcement values is detected (for instance, sometimes very low and sometimes very high). In this case, we can suppose that the perceptual state p is ambiguous (that is, it represents more than one real state with different values of future reinforcement) and leads to a perceptual aliasing. To avoid the problem of choosing the right action for this perceptual state p, *the algorithm develops a policy that avoids to pass through this aliased perceptual state* p while solving the problem. This policy is achieved by returning a very negative immediate reinforcement value when the state p is perceived. In this way, behaviors that solve the task avoiding ambiguous states are learned. The `lion algorithm` [206], and the `Consistent Representation` [207] are examples of this approach.

Unfortunately, it is not possible, in some cases, to solve the task without passing through aliased states; see, for instance, the McCallum's mazes 4.4.1.

4.2.3 Disambiguate states by learning a world model

Another approach to solve the perceptual aliasing problem consists in exploiting the following idea: If the problem consists in the ignorance of the current real state, caused by a limited perceptual system, then the solution could consist in learning a world model to help the agent in determining the current real state from the partial information provided by its perceptual system.

Learning a world model usually consists in assuming a Markovian environment where each perceptual state corresponds to a different real state. When an incoherence with this assumption, concerning to a perceptual state, is found (for instance, when the expected return from a perceptual state is variable), the algorithm assumes that the perceptual state is aliased and represents two different real states. In this case, a splitting in the world model of the perceptual aliased state, is performed. From this new model, the algorithm tries to learn a new evaluation of the states, finding and solving new incoherences (repeating the described procedure) until the desired policy is found.

One example of this approach is McCallum's `Utile Distinction Memory` (`UDM`) [119], [121], where a graph that represents the world model is learned from the observation of a non-Markovian behavior in some perceptual states of the agent (in terms of the estimations of future discounted reinforced obtained).

One of the main disadvantages of this approach is that it is usually very expensive in the number of experiences needed to learn the model. That means that this approach is only useful for modest domains. Another problem is that the learned model is useful *only* for the specific environment in which the agent has learned the model, but it is not useful for learning general behaviors (as, for instance, in the blocks world case; see section 3.2).

It is interesting to note that this approach is closely related to some approaches to the *generalization* problem in reinforcement learning, for instance the `G-algorithm` [38] and the `Parti-Game` [137]. These approaches start assuming maximal generalizations for all states. When a generalization shows a non-Markovian behavior, it is specialized. The algorithm finishes with a set of generalizations that covers all states and does not produce ambiguities about the action to choose while solving the task.

The close link between the disambiguation-by-learning-a-model and the generalization approaches amounts to the fact that *any limitation of perceptual information of the domain implies a forced generalization of real states*. In the case of generalization, when a non-Markovian generalization is detected, new information is added to disambiguate the aliased states (all information is available but used only when it is needed in the specialization process). In the case of perceptual aliasing, the new information is not available to the perceptual system.

4.2.4 Belief vectors

Another different approach is generically named *belief vectors*. In this case, the agent does not learn by means of a search of the policy in the perceptual space, but in a space of *beliefs*.

The belief space is constituted by belief vectors. A belief vector consists in a probabilistic weighting of each real state of the environment that expresses the agent's belief of being in them. For instance, if there are 4 possible *real states* (not perceptual states) in a small domain, a belief vector in that domain would be represented as $< p_1, p_2, p_3, p_4 >$, where p_1 is a probabilistic value that represents the agent's belief of being in the state s_1, etc.

In this approach, the agent tries to find a policy from an initial belief vector (usually with an equal probability for each state), and a *world model* that is useful to calculate which is the probability value of being in a determinate real state when an action is executed from another real state and a determined perceptual information is obtained.

The initial belief vector is modified depending on the executed actions and the obtained perceptions, becoming a summary of the agent's perception-action history while solving the problem. Often, the agent learns to solve the problem (when it used an appropriate reinforcement learning algorithm) by executing actions that disambiguate the real state in which the agent is, obtaining a belief vector in which all p_j are zero except in p_i where the probability value is 1.

This approach is useful when the agent suffers a perceptual aliasing in its interaction with the environment and *a world model is available to the agent*. In this case, the algorithms can find optimal policies considering the perceptual aliasing. Examples of this approach are the `Witness` algorithm [33], [97] and `SPOVA` [145].

The main disadvantages of this approach are that the algorithms are very expensive (although new and more quickly algorithms appear continuously) and that *they need an a priori knowledge of the environment* (the number of real states and a world model). Moreover, because a world model is needed and the algorithms are oriented to learn optimal policies, this approach is not adequate to learn general behaviors.

Other approaches based on belief vectors, as the `Perceptual Distinction Approach` [40], try to learn not only the policy but also the world model. The learning of the world model is based in a splitting technique similar to the used in the previous section when a non-Markovian perceptual state is found. Nevertheless, this algorithm does not guarantee an optimal policy. Also, unfortunately, the learning of the world model is very expensive in terms of the necessary number of experiences in order to learn the behavior.

4.2.5 Automatic Programming

Another approach consists in automatically produce, by means of learning methods, programs that describe an appropriate behavior to solve the task.

Perceptual aliasing is solved in this approach because the different locations in the program where an ambiguous perception appears, allow to distinguish the appropriate action to execute. That is, *equal perceptions can have defined different responses when they appear in different points of the program.* This is the usual way to solve the perceptual aliasing problem in *human programmed* behaviors for reactive systems. For instance, the following program returns different actions to execute (`a` and `b`) for the same perception (`p`) depending on the location of the program where the aliased perception appears.

```
for i:=1 to n
begin
 s:= perceive;    {* sets to s the current perceptual state *}
 if s = p
     then execute_action(a)    {* execute the ``a´´ action  *}
     else execute_action(c);   {* execute the ``c´´ action  *}
 s:= perceive;
 if s = p
     then execute_action(b)    {* execute the ``b´´ action  *}
     else execute_action(c);   {* execute the ``c´´ action  *}
end;
```

This approach is very different from the usual way of learning policies with classical reinforcement learning algorithms. In this case, the different possible situations of the environment are not valued depending on the future reinforcement to obtain. Instead of that, this approach performs a *search in the space of possible programs* in order to find the adequate program.

One example of this approach is the conjunction of `EIRA` and `Levin Search` proposed in [209]. Other examples are the methods based in the using of `Genetic Programming` techniques [85].

Unfortunately, it seems that this kind of techniques need more experiences than other techniques in order to learn an adequate behavior (see, for instance, results obtained in section 4.4.2). However, it is important to note that this approach can solve the perceptual aliasing problem.

4.2.6 Hierarchical approach

Hierarchical learning usually consists in decomposing a task into simpler sub-behaviors. Each sub-behavior is learned separately and, finally, all sub-behaviors are coordinated into a new behavior that can solve the complex task (see section 5.3 for a more extensive review of this approach). This approach is generally used to scale up reinforcement learning mechanisms enough to solve tasks with a long delayed reinforcement.

An interesting side-feature of hierarchical learning is that it can solve the perceptual aliasing problem. For instance, let us assume that there are two different real states that (due to a limited perception) are perceived as the same one, and that for these two real states, different actions must be executed in order to solve the task. In this case, the perceptual aliasing could be solved if the two real states were treated in different behaviors. One behavior would determine the first action for the aliased perception and, the other behavior would determine for the same perception the second action. Thus, when one perceptual state needs to take two different actions, the perceptual aliasing could be solved by learning two different behaviors.

Instances of this approach to solve the perceptual aliasing problem are the Ring's works [155] [154] and the Wiering and Schmidhuber's work [208]. In these cases, the hierarchical approach is not used to speed up the learning of a complex task. They only take advantage of the side-effect of hierarchical learning in order to build algorithms able to solve the perceptual aliasing.

4.2.7 State generation approaches

Another interesting approach is the `GTRL-S` algorithm proposed by Kaelbling in [83, Chapter 8]. It consists in an architecture that builds (in a *generate and test* cycle) boolean circuits that calculate the adequate action to execute for a given perception. The architecture allows to maintain, by means of *flip-flop* circuits, information about arbitrarily long past situations. Those *flip-flop* circuits produce an internal information (internal state) that can disambiguate the aliased states.

Unfortunately, experiments seem to show that it is very expensive to learn this boolean circuits, and that it is difficult to obtain results in tasks with a delayed reinforcement. Usually, the only way apply this approach in complex task is by supplying an adequate bias over the search process in the space of boolean circuits.

4.2.8 Memory approaches

Finally, another interesting approach to solve the perceptual aliasing problem consists in memorizing information about the past actions and/or perceptions experienced by the agent in the process of solving the task. Information about past perceptions, stored in a memory device, could be useful in order to disambiguate the current perceptual aliased state.

The stored trace of actions-perceptions used to disambiguate states could be (in a extreme position) all the agent's sensory-motor history, but this gives an excessive amount of information that becomes difficult to manage. In most cases, *it is not necessary to store the complete history of the agent, but only a selected part of it.* We classify the memory approaches depending on the selection procedure used.

Continuous window of the last states and/or actions

This approach considers that the last actions or situations are the most adequate for disambiguating a perceptual state. Thus, a window of length t is used to store the last t actions and/or situations. The content of the window will be used to complement the information provided by the sensory system in order to choose the action to be executed.

One example of this approach is Lin's work in the `window-Q architecture` [93]. In this work, he studies the performance of a neural network that uses a window of the t past perceptions in order to learn the adequate `Q` function to solve a problem. He tested this architecture, obtaining acceptable results. The length t of the window has to be determined beforehand as a fixed parameter for the task. This is a crucial parameter because, if the length is too small, the problem cannot be solved, but if it is too large, the problem becomes too difficult to be learned. Usually, this parameter is found by *trial-and-error*.

This approach was used in the previous chapter to solve the `searching for a block` task (see section 3.2). In that case, the length of the window t was set to 1, and the stored value was only the last action executed.

Other examples of the memory approach try to determine the length of the window dynamically, while learning takes place. For instance, the McCallum's learning algorithms `Suffix Tree` [126] [127], `USM` [125] and `NSM` [123] store traces of different length depending on the case. The length of the window grows as the benefits from adding more historic information can be obtained. Another example is Ring's `Temporal Transition Hierarchies` [155] that builds a hierarchy of behaviors by using second order neural networks. In this algorithm, if a window of length t is needed to disambiguate a state, then a hierarchy of behaviors (of maximum t levels) is built to solve the problem.

Obviously, in this approach, the task becomes more complex to solve by using this algorithms as the length t of the window needed to solve the problem grows.

Both approaches, fixed an variable length window, *assume that all last t actions or perceptions are useful to solve the ambiguity problem.* This is not generally true.

It could happen that the relevant information to clear up the current ambiguous perception were only the action/perception executed/perceived t steps ago (t being arbitrarily large), but no the others executed/perceived from it to the most recent one. In this case, the information stored from location $t-1$ to location 1 of the window is not useful to disambiguate the current perception; only information stored in location t becomes useful for this purpose. If t is large enough, the learning algorithm must deal with a huge amount of irrelevant information. This is one of the main reasons for developing a more selective approach (see the active memorization section 4.3).

Recurrent Neural Networks

Another approach is based on the use of recurrent neural networks [64]. Neural networks of this kind disambiguate the perceptual aliased state with additional information stored in `context units`. Context units vary their content as a function of their own values and the current perceptual values. This function is learned at the same time that the policy. Context units store *a resume of all the past history of the agent* and, theoretically, can retain information from arbitrarily long past situations. This is a feature that distinguishes the window's approach from the recurrent neural network's approach. On the other hand, recurrent neural networks have to pay for this feature an increment (compared with the fixed window approach) in the number of trials needed to learn to keep the relevant information.

In Lin's work [93] the `recurrent Q-architecture` (that integrates recurrent neural networks with `Q-learning`), is presented.

Unfortunately, it is difficult for recurrent networks to learn the influence of a long time past state over the current one. This difficulty seems to increase excessively with the time distance that separates the relevant information from the current situation.

4.3 Active Memorization

In this section, we describe our approach to solve the perceptual aliasing problem. This new approach is called *active memorization* and consists in using memory registers and extending the repertory of actions of the agent with actions that actuate over those memory registers. The memory registers will be used to disambiguate the aliased states that prevent the learning of an adequate behavior. The memory actions will be used by the agent, when it finds it suitable, to update the memory registers. Memory actions will not have, for the agent, a different treatment than usual actions: they will be chosen in competence with them and, when reinforcement is obtained, the learning process will update their statistics in the same way that statistics of usual actions are updated.

In order to differentiate, when necessary, memory actions from the usual actions, we will call the former *internal actions*, and the latter *external actions*.

The main difference of this approach with respect to the previously described memory-based ones, lies in the different *selective process* used to store relevant past information. *The idea behind this new approach is that the agent will not only learn how to solve a problem with external actions, but also to collect the needed information to disambiguate the aliased states that prevent the learning of an adequate policy.* We call this approach *active memorization* because the agent has to execute deliberately an action in order to store the relevant information, in opposition to other memory approaches in which memorization is done in a passive way (automatically).

Active memorization is a lot more selective approach than the other ones based on memory as, for instance, the window's approach in which all last t actions executed and/or perceptions experienced by the agent are stored. In particular, our approach does not assume the continuum hypothesis (that all last t situations are relevant for solving the perceptual aliasing problem).

Compared with other approaches that are not based on memory, it is interesting to note that our approach can solve the perceptual aliasing problem avoiding some constraints. For instance, it is possible to pass through aliased states without risk (this was not possible in the *ignoring* and the *avoiding aliased states* approaches; see sections 4.2.1 and 4.2.2). It is possible to learn a behavior without having a model of the environment (in the *belief vectors* approach a world model is necessary; see section 4.2.4). Finally, in our approach, to learn a world model of the environment is not necessary (in the *disambiguate states by learning a model* approach this is necessary; see section 4.2.3).

We must remark that our approach, by being independent of a world model, allows learning general behaviors. Our approach is, then, an adequate extension to the CANDID algorithm.

In the next section, we will study previous works related to this approach. Later, we will describe two different architectures based on our *active memorization* proposal. Finally, in section 4.4, an experimental test for the proposed architectures will be performed.

4.3.1 Antecedents

The works of Littman [95], Wilson [211], and Cliff and Ross [46] are (to the best of our knowledge) the sole antecedents of the `Active Memorization` approach.

Littman's work

In Littman's work [95] about *memoryless policies* (that is, policies that do not make use of memory), we can find the first mention that we are aware about using actions to memorize. In that paper, firstly, Littman showed that finding deterministic *optimal* policies is an NP-complete problem and, secondly, he developed a heuristic Branch and Bound algorithm to find them. Later, when faced with the b McCallum's maze (see figure 4.3 and section 4.4.1), Littman confirmed that a memoryless deterministic policy that could solve this maze does not exist. Then, he built an agent with an elementary memory consisting of one bit register. In order to update this memory bit, each action of the agent was duplicated (e.g. from action a to a_0 and a_1). The two new actions produce the same external effect as the original action a, but one of them (a_1) sets on the memory bit and the other one (a_0) sets it off. With this memory bit and these new actions, the agent became able to disambiguate the ambiguous perceptual states of the b McCallum's maze.

Littman's idea can be generalized from one memory bit to r memory bits. In order to do this, for each possible memory configuration of r bits and for each external action of the agent, a new action that actuates as the external action and that sets the memory bits to that configuration, is created. An instance of these new actions could be action $a_{10\ldots1}$ that produces the same external effect than the original action a and that sets on the first memory bit, off the second, ... and on the *r-nth*.
In this case, the total number of actions of the agent is:

$$Actions = n\,2^r \tag{4.1}$$

where n is the number of external actions of the agent. As we can see, the number of actions available to the agent, grows exponentially with the number of memory registers.

On the other hand, each perceptual state is complemented with the information stored in memory registers. This generates the following number of different states the agent can come across:

$$States = s\,2^r \tag{4.2}$$

where s is the number of different perceptual states that can be perceived by the agent's sensory system.

In case that each memory registers were extended from two possible values to c_i possible values, the number of actions and the number of states would respectively be:

$$Actions = n \prod_{i=1}^{r} c_i \tag{4.3}$$

$$States = s \prod_{i=1}^{r} c_i \tag{4.4}$$

where s is the number of different perceptual states that can be perceived by the agent's sensory system, r is the number of memory registers, and c_i is the number of different values that the *i-nth* memory register can take.

Unfortunately, this approach has not been developed for the usual reinforcement algorithms, maybe because Littman thought that this approach produces a combinatorial explosion in the number of actions with the growing number of memory bits (as he stated in a posterior publication [96]). His conclusions [95] seemed to tend towards the stochastic approach to solve the aliasing problem.

We think that this a pessimistic view about the potentials of this approach. We describe below a new architecture that uses actions for memorizing and that avoids the exponential increase of the number of actions available to the agent.

ZCS with temporary memory

In the area of *classifier systems*, there has been another attempt to develop the idea of active memorization [46]. In this work, the `Zeroth-level Classifier System` (`ZCS`) described by Wilson in [211] was extended with a *temporary memory* (proposed also by Wilson in the same paper). This temporary memory was represented by means of binary registers. These registers were used to complement the perceptual information of the agent in order to select the most appropriate classifier to be activated. This approach is called `ZCSM`, where `M` takes the value of the number of memory registers used.

Each classifier has two parts: a *matching part* and an *action part*. The matching part defines a combination of possible sensory values and a combination of memory register values (in both cases, the *do not care* value "#" can be used). When the matching part of a classifier matches the current situation (that is, when it matches the current perception and the memory registers setting), the classifier is selected. When one of the selected classifiers is activated, the agent executes the action part of the classifier. The action part defines the external action to execute in the current situation and the procedure to modify *each* memory register. In this case, there are three ways to modify each register: `set`, `reset` and `leave-alone`.

In this approach, the number of possible action parts that can be defined in a classifier is:

$$Actions = n\, 3^r \tag{4.5}$$

where n is the number of external actions of the agent and r is the number of memory registers used. As we can see, the number of actions available to the agent, also grows exponentially with the number of memory registers (as in the Littman's case).

On the other hand, the number of possible states the population of classifiers has to cover in the `ZCSM` approach is:

$$States = s\,2^r \tag{4.6}$$

where s is the number of different perceptual states that can be perceived by the agent's sensory system and r is the number of memory registers used.

In case that each memory register were extended from two values to c_i values, the number of actions and the number of states would respectively be:

$$Actions = n\prod_{i=1}^{r}(c_i + 1) \tag{4.7}$$

$$States = s\prod_{i=1}^{r} c_i \tag{4.8}$$

where n is the number of external actions of the agent, s is the number of different perceptual states that can be perceived by the agent's sensory system, and r is the number of memory registers used.

As it can be seen, the `ZCSM` approach is equivalent to the Littman's approach but for classifier systems. In [46], the `ZCSM` approach was tested in several POMDP maze problems, obtaining acceptable results. Despite that fact, in the same paper the extensibility of the developed approach was put under question. The authors foresighted a combinatorial explosion in the number of actions when the number of memory bits grows. To mitigate that problem, they also proposed as a partial remedy to the combinatorial problem, the separation of external actions and memory actions (this was the Wilson's original proposal in [211], but not used in the `ZCSM` mainly to improve the genetic *cross-over* operator).

In spite of this, the authors did not develop their new proposal and neither did they test it, in part because the exponential problem is still present, in part because they found that the stochastic method use by `ZCS` to determine which action to execute, was enough to solve the maze problems without the help of memory bits (a similar conclusion to the one proposed by Littman). We think that the stochastic approach is not an appropriate way to deal with the perceptual aliasing problem (see section 4.2.1).

Let us analyze this second proposal that we call `ZCSM II`. In this case, the number of possible action parts that can be defined used in a classifier is:

$$Actions = n + 3^r \tag{4.9}$$

where n is the number of external actions of the agent, and r is the number of memory registers used. In this case, the number of actions of the agent grows exponentially as in the previous approaches.

On the other hand, the number of possible states the population of classifiers has to cover in the `ZCSM II` approach is:

$$States = s\,2^r \tag{4.10}$$

where s is the number of perceptual states provided by the agent's sensory system and r is the number of memory registers used.

In case that each memory register is extended from two values to c_i values, the number of actions and the number of states would be, respectively:

$$Actions = n + \prod_{i=1}^{r}(c_i + 1) \tag{4.11}$$

$$States = s \prod_{i=1}^{r} c_i \tag{4.12}$$

where s is the number of perceptual states provided by the agent's sensory system, n is the number of external actions of the agent, and r is the number of memory registers used.

4.3.2 New Active Memorization Models

One of the problems of Littman's and `ZCS` approaches is the exponential growth in the number of actions available to the agent. This growth causes an increase in the number of possible decisions that the agent could take at each decision step (each action becomes a possible decision). This is a very important problem because the performance of any *trial-and-error* algorithm, under those circumstances, will clearly decay.

Our proposal consists in relaxing the influence of each action on the memory registers. This is achieved in the following way:

1. The external and the internal actions are separated, as in the `ZCSM II` approach. This reduces the number of actions but it does not prevent its exponential growth.

2. The number of internal actions is reduced to *only one* action for each memory bit, that sets it to the complement of the current value. This prevents the exponential growth in the number of actions.

Someone could think that these relaxations in the architecture could decrease the power of the resulting approach for learning a behavior under perceptual aliasing. This could be the price to pay for making the approach computationally tractable.

Fortunately, not all the power of the initial architectures (the ones proposed by Littman and ZCS) is usually needed. Moreover, it becomes usually excessive, producing a lot of possible solutions that, in fact, are equivalent (because they become the same solution with an appropriate change-of-variable, for instance, by substituting the number of one memory bit for another one or by replacing the value of a memory bit for its complement.)

In spite of this, there could exist problems that the initial architecture could solve with k memory registers but that our architecture could not solve using the same number of memory registers. We distinguish two cases:

- *The new architecture cannot solve the task with k memory registers.* In this case, the solution consists in augmenting the number of memory bits. As the new approach does not show an exponential increase in the number of actions, usually it will be better to use the new architecture with more memory bits than using the initial architecture.

- *The new architecture cannot solve the problem because it is necessary to carry out an external action and to change a memory register at the same time.* In the initial architecture, the actions of the agent could, in a single time step, carry out an external action and change the memory registers. In our approach, as the external and internal actions are separated, this is not possible. Nevertheless, to change the memory registers and to carry out an external action at the same time becomes only *necessary* in cases where the execution of an external action does not change the perceptual input of the agent. A possible solution to these cases could be to use the second type of architecture that we propose below (see section 4.3.2).

On the other hand, Wiering and Schmidhuber indicated in [208] that a possible problem of the active memorization approach can appear when, periodically, the agent takes non-policy actions. Under these circumstances, it is difficult to maintain the value of a memory bit for a long time. *As the number of steps grows, the possibility of taking a non-policy action that erases the stored memory, also grows.* This problem is increased when *each action execution* (as in the Littman's and ZCSM approaches) modifies the memory bits .

These observations support the above decision about separating external from internal actions. Nevertheless, we must note that in our case, the problem suggested in [208] is not present when active memorization is used together with the CANDID learning algorithm because it never takes non-policy actions (see section 3.5.1).

In the following sections, we will develop two different architectures that implement all these ideas.

Active Memorization I

In this first model, the agent's perceptual system is complemented with an adequate set of memory registers. *For each memory register mr_i, there is a memory action ma_i that updates it to its complementary value.* This means that the ma_i action sets the mr_i memory register on when it is off, and sets it off when it is on.

Usually, the memory registers have only two values, but we could extend the approach to c_i*-valued* memory registers. In this case, for each memory register mr_i, there is a memory action ma_i that adds 1 module c_i to it.

In this case, the total number of actions available to the agent is:

$$Actions = n + r \tag{4.13}$$

where n is the number of external actions of the agent and r is the number of memory registers. *As we can see, the number of actions available to the agent, does not grow exponentially with the number of memory registers, as it happened in the previous approaches.*

On the other hand, each perceptual state is complemented with the information stored in the memory registers. This generates the following number of different states the agent can come across:

$$States = s \prod_{i=1}^{r} c_i \tag{4.14}$$

where s is the number of perceptual states that can be perceived by the agent's sensory system, c_i the number of different values that the mr_i register can take, and r the number of memory registers used.

Active Memorization II

In some cases, the lack of perceptual information about the environment is so extreme that, although the agent executes successfully the adequate external action for a perceptual state, the perceptual information that the agent obtains from the environment does not change. This becomes problematic because, when the agent has to choose the next action to be executed, it will choose the same action that was executed in the last step (because the perceptual state remains the same) *without noticing that the action already has been executed!* As memory actions are separated from external actions in our approach, there is no way to solve this problem.

In these cases, in order to solve the task, the agent should execute different actions with the same perceptual state, which means that the perceptual aliasing problem is not solved. It could only be solved with active memorization if the execution of each action in any decision step could change the state considered by the agent (perceptual information plus internal state). This effect can be produced by a new mechanism that we propose, called *oscillation*.

The oscillation mechanism consists in a special state register that automatically changes its value after the execution of one action. The value of the oscillator register is considered as constituent of the state (as the perceptual information and the memory registers are) in order to choose the next action to be executed. *In this way, although the execution of one action did not change the perceptual input, the state of the agent has been modified because the value of the oscillator has changed automatically.*

From the resources point of view, in addition to memory registers, an oscillator register will be used in this new model. The number of values the oscillator can take depends on the number of different actions the agent has to take without a change in the perceptual state. *The oscillator register changes its value at each decision step adding 1 to the register counter module c_0, where c_0 represents the maximum number of values the oscillator can take.* It is desirable to use the minimum c_0 value necessary to solve a problem, because it influences the number of states and the length of the chain solution.

Two values should be enough for most tasks: after the execution of one external action the internal state changes (because the oscillator value has changed), then the agent could execute a memory action that changes the state of the agent (and returns the oscillator to the original value). In this case, when the oscillator returns to the initial value, the state of the agent is different than the resulted from the action execution, and the agent can choose another action for execution. Nevertheless, in some cases, a higher number of values for the oscillator is necessary. The memory actions that we have defined in this approach are a very simple form of memory. Each memory action only changes one memory bit by setting it to its complementary state (or by adding one to it). This implies that for changing adequately the internal state of the agent it could be necessary to execute more than one memory action. In this case, we would need so oscillator values as memory actions must be executed in order to change adequately the internal state of the agent, plus one. On the contrary, we could avoid this problem by making the memory actions more powerful, but this would return us to the combinatory problem. However, in the experiences in which this approach was tested, all problems could be solved with a maximum of $c_0 = 2$.

In this new model, the total number of actions available to the agent is:

$$Actions = n + r \tag{4.15}$$

where n is the number of external actions of the agent and r is the number of memory registers used. The number of actions remains the same as in the previous approach, because the oscillator value changes automatically. In spite of this, *sometimes*, the performance of the learning algorithm could be improved by adding a `rest` action that does not do anything[4].

On the other hand, each perceptual state, complemented with the information stored in the memory registers, is extended with the value of the oscillator register. This generates the following number of different states the agent could come across:

$$States = s\, c_0 \prod_{i=1}^{r} c_i \tag{4.16}$$

where s is the number of perceptual states provided by the agent's sensory system, c_0 is the maximum number of values the oscillator can take, c_i the maximum number of different values that the mr_i register can take, and r the number of memory registers used.

4.3.3 Comparison

In this section, a comparative evaluation of the different approaches to active memorization will be made.

Tables 4.1 and 4.2 summarizes the analysis of the different approaches that we have made in the previous sections. Each table shows for each approach the values for some features that will be used in order to perform the comparison. These features are selected in order to measures the size of the space to be explored while learning a behavior. We must note that, generally, the smaller the space to be explored for learning, the smaller the time spent to find the adequate behavior. The features selected are the following:

- The *number of states* formed with both perceptual and memory information. It shows the size of the policy table to learn. This number is calculated by the mathematical product of the number of different perceptual states that can be perceived by the agent's sensory system s (that is supposed equal for each approach) and the number of different configurations that are possible to obtain with the memory device chosen (window or registers).

- The *number of actions* the agent can use at each decision step, including external and internal actions. It represents the branching factor of the search space. There are three qualitatively different possibilities for this feature: no more actions than the external ones (for instance the window approach, where there are no actions to modify the memory device; it is automatically updated), multiplication of actions (Littman and `ZCSM`s approaches, where

[4] We must note that when this action is executed (as when any other action is executed) the value of the oscillator is automatically updated.

Approach	Num. states	Num. actions	Number of statistics
Window's approach	$s\,2^w$	n	$s\,n\,2^w$
Littman's approach	$s\prod_{i=1}^{r} c_i$	$n\prod_{i=1}^{r} c_i$	$s\,n\,(\prod_{i=1}^{r} c_i)^2$
ZCSM	$s\prod_{i=1}^{r} c_i$	$n\prod_{i=1}^{r}(c_i+1)$	$(n\prod_{i=1}^{r}(c_i+1))\,s\prod_{i=1}^{r} c_i$
ZCSM II	$s\prod_{i=1}^{r} c_i$	$n+\prod_{i=1}^{r}(c_i+1)$	$(n+\prod_{i=1}^{r}(c_i+1))\,s\prod_{i=1}^{r} c_i$
Active Memorization I	$s\prod_{i=1}^{r} c_i$	$n+r$	$(n+r)\,s\prod_{i=1}^{r} c_i$
Active Memorization II	$s\,c_0\prod_{i=1}^{r} c_i$	$n+r$	$(n+r)\,s\,c_0\prod_{i=1}^{r} c_i$

Table 4.1: Summary table I. Where s is the number of perceptual states, n is the number of external actions, w is the length of the window measured in bits, c_i is the number of possible values for memory i, r is the number of memory registers, and c_0 is the number of possible values of the oscillator.

each external action has also associated an internal action to modifies the memory registers, and/or there is one action for each possible configuration of the memory registers), and addition of actions (for instance our approach, where external actions and internal actions are separated and also there is only one action for each memory register).

- The *number of statistics* represents the amount of statistics that must be stored in the table used by the CANDID algorithm for learning a behavior. This indicates the spatial resources needed for each approach. For each state and each action there is one statistic S_A and one F_A. Thus, this value can be calculated by the mathematical product of the number of states and the number of actions.

- The *number of policies* shows the number of different policies tables that could be built from the set of actions and states. Each possible policy describes a behavior, and the policy that solves the problem (if it exists) is one of them (although it is not always the only one). Thus, this number indicates the size of the space in which the behavior solution has to be be search and it usually becomes an index of the complexity of the task to be learned. As each policy consists in choosing one action for each state, the number of policies can be calculated as the number of actions powered to the number of states.

Approach	Number of policies	Length of solution
Window's approach	$n^{s\,2^w}$	k
Littman approach	$(n \prod_{i=1}^{r} c_i)^{s \prod_{i=1}^{r} c_i}$	k
ZCSM	$(n \prod_{i=1}^{r} (c_i + 1))^{s \prod_{i=1}^{r} c_i}$	k
ZCSM II	$(n + \prod_{i=1}^{r} (c_i + 1))^{s \prod_{i=1}^{r} c_i}$	$k + m$
Active Memorization I	$(n + r)^{s \prod_{i=1}^{r} c_i}$	$k + m$
Active Memorization II	$(n + r)^{s\, c_0 \prod_{i=1}^{r} c_i}$	$k + m$

Table 4.2: Summary table II. Where s is the number of perceptual states, n is the number of external actions, w is the length of the window measured in bits, c_i is the number of possible values for memory i, r is the number of memory registers, c_0 is the number of possible values of the oscillator, and m is the number of memory actions needed in each approach in order to solve the problem (see text).

- The *length of the solution path* indicates the number of steps needed to solve one experience compared with the number of steps k needed in Littman's approach. This feature only expresses that if the Littman's approach solves the problem with k steps, the other approach can also solve it with that number of steps. That value is, of course, difficult to calculate, and *the value shown in the table only indicates that whether more or the same steps than in the Littman's approach are needed.* This value is useful also to give an idea of the complexity of the task because the longer is the sequence of actions that solves the problem, the more difficult is to find it (see section 2.2.1).

From the comparison, the following conclusions can be drawn:

1. The number of states is the same in all approaches except in the window approach and the `Active Memorization II`. Unlike the other approaches (that learn to select the relevant information), the window approach has to maintain a long trace of past actions in order to be sure that it stores the relevant information. Thus, *the length of the window w used in the window approach is usually a lot larger than the number of memory bits r used in the other approaches.* This means that the window approach needs a lot more bits of information in order to disambiguate the perceptual aliased states. This becomes a problem because, in the window approach, the number of states increases exponentially with the length of the window.

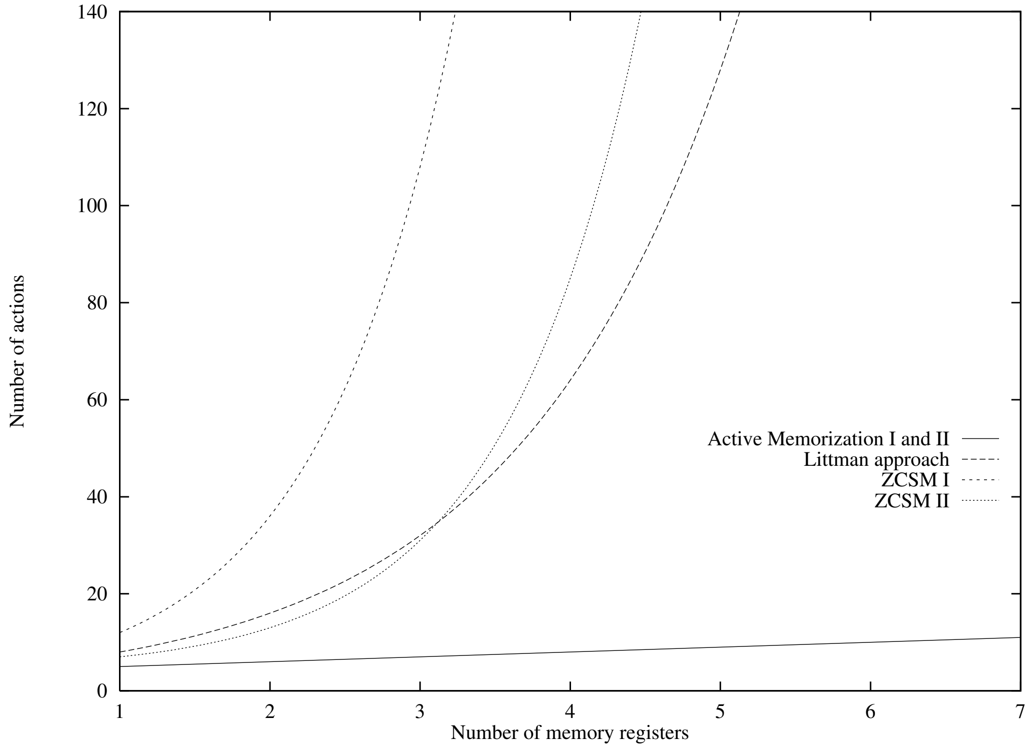


Figure 4.1: Increase of the number of actions for each approach as the number of register memory grows. The number of original actions is set to 4.

This exponential growth makes the window approach unfeasible when the length of the window needed becomes too large. On the other hand, in the `Active Memorization II` approach, a multiplying factor c_0 appears. Taking into account that this approach is only used in case of a extreme lack of perceptual information, and that the coefficient c_0 does not implies an exponential growth in the number of states, we can conclude that this approach (in this aspect) is assumable.

2. The number of actions needed by the active memorization approach is really very low compared to other approaches (except the window approach). This difference is increased in a exponential way as the number of memory registers grows. This can be seen in figure 4.1.

   On the other hand, comparison of the active memorization approach and the window approach shows that the difference in the number of actions is only the same that the number of memory registers used (that usually is very low).

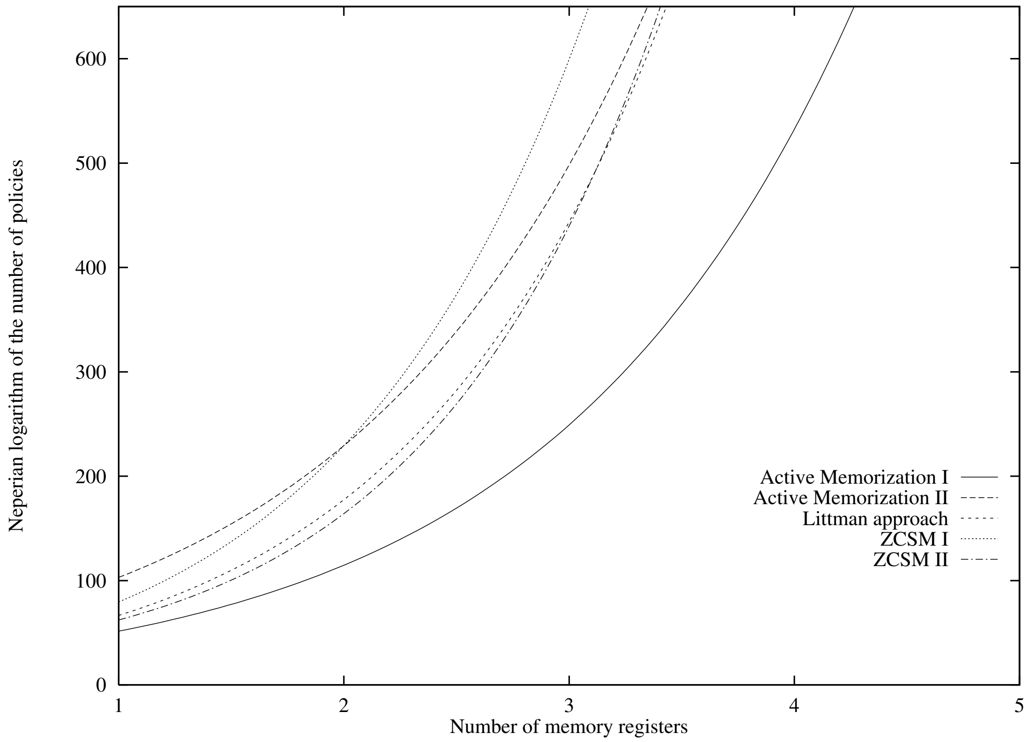


Figure 4.2: Increase of number of policies for each approach as the number of register memory grows. The number of original actions is set to 4 and the number of perceptual states 16. Policies scale is logarithmic.

3. About the number of policies, it can be noted that the exponential order of each approach is the same (except for the window approach and the `Active Memorization II`). The main difference consists in the exponential base of each approach. The lower base corresponds to the `Active Memorization I` approach. The growth in the number of policies versus the growth in the number of memory registers, for each approach (except the window approach) can be seen in figure 4.2.

   The windows approach has an exponential order that is usually higher than the other ones because, as commented when the number of actions for each approach were compared, the length of the window w is usually a lot larger than the number of memory bits r used in the other approaches.

   Finally, in the `Active Memorization II` approach, the multiplying factor c_0 is needed to solve the case of an extreme lack of information [5]. This

[5] That is, when after the successful execution of one action, the sensory input has not changed.

means that it is used only when other approaches cannot be used, and then it cannot be compared with them. Nevertheless, in the figure can be seen that from 4 memory registers on, it becomes the second best approach in the number of policies.

4.4 Test Domains

The active memorization approach has been tested in different well-known domains. It is important to note that in the kind of problems selected we are not interested in obtaining optimal solutions but only acceptable ones. The tested environments are, then, problems not about optimization, but about learning acceptable general behaviors.

In the following sections, the obtained results in these domains are described. The domains have been classified depending on the degree of perceptual aliasing they present.

4.4.1 Limited perception I

In this section, we test the *active memorization* approach in some environments that suffer a moderate perceptual aliasing. Problems in these environments can be solved by using the `Active Memorization I` architecture.

McCallum's mazes [123], [125]

The McCallum's mazes (see figure 4.3) is a set of 5 labyrinths designed by McCallum in order to test the `UDM` [119], `USM` [125] and `NSM` [123] algorithms. The *b* maze has been referenced in several other works, for instance, [46], [95] [97], and [155].

Each environment consists of a maze with a goal position (marked with the "G" letter). These mazes are similar to those used by Sutton [181] and other researchers to show the abilities of their reinforcement learning algorithms. The main difference is that in McCallum's mazes, the information that the agent gets about the environment is limited by its perceptual system. In this case, the agent perceives only the presence or absence of a wall in the immediate *North*, *East*, *West* and *South* positions. The agent has 4 actions to move around the maze: `Move-North`, `Move-East`, `Move-South` and `Move-West`.

The characteristics of each maze are shown in table 4.3. There, the following informations are collected: the name of the maze, the number of different states with complete information, the number of perceptual states with the agent's perceptual system, and the number of actions originally available to the agent. All these parameters depends on the maze and are equal for any approach that tries to solve the maze. In the same table table the characteristics of the task when

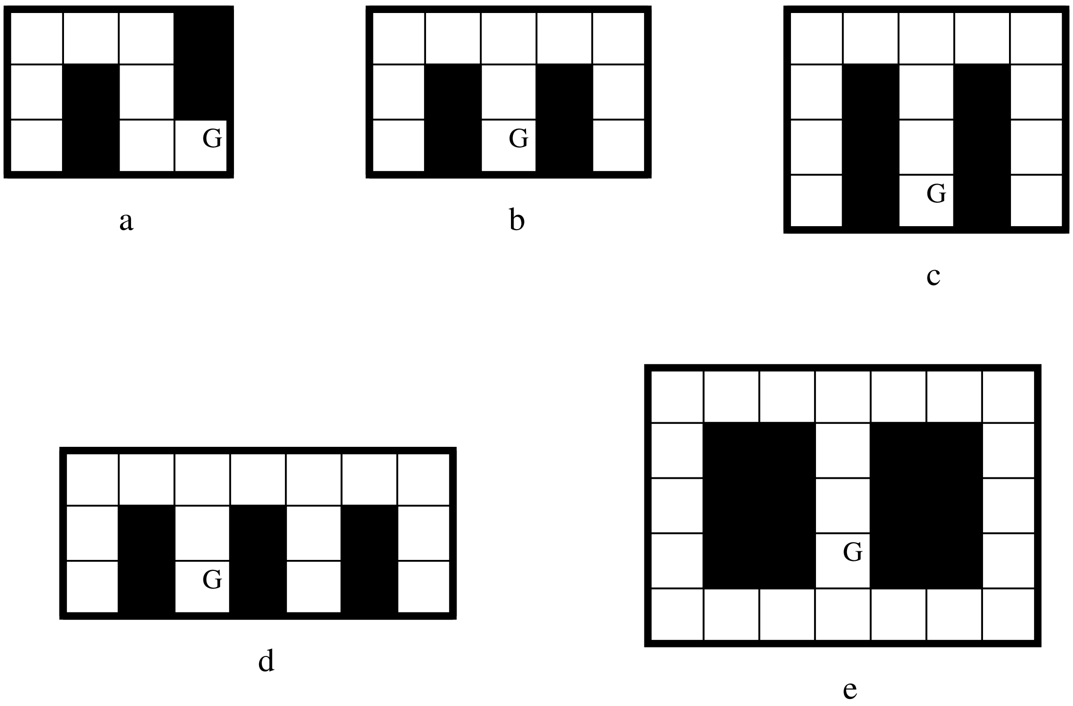


Figure 4.3: McCallum's mazes (a,b,c,d) [123], (e) [125].

the CANDID learning algorithm is used with `Active Memorization I`, are also shown. The table shows the number of memory bits used to solve the problem, the number of actions available to the agent by using this approach, the number of different states including the memory information, and finally the number of possible policies under these circumstances. This last information is indicative of the complexity of each maze[6]. In Markovian tasks, the number of states represents an adequate estimation of the complexity of the task, but in tasks with perceptual aliasing, the real states become aliased and, thus, the number of states become a not adequate index for measuring the complexity of this kind of tasks. In order to express the complexity of these tasks, we prefer to use the number of possible policies. It is important to note that the number of possible policies depends on the method used to solve perceptual aliasing, and that the presented value is for the `Active Memorization I` approach (other approaches present other values, usually higher; see section 4.3.3).

In the original version of the mazes proposed by McCallum, perceptions

[6]We must remark that this value is *only* indicative. For this reason, although almost all mazes present the same number of possible policies, the obtained results for each maze are different.

Maze	*States*	*Percep. states*	*Number actions*	Mem. bits	Actions used	States	Number of policies
a	7	6	4	1	5	12	$5^{12} \approx 2.4\,10^8$
b	10	6	4	1	5	12	$5^{12} \approx 2.4\,10^8$
c	13	6	4	1	5	12	$5^{12} \approx 2.4\,10^8$
d	14	6	4	1	5	12	$5^{12} \approx 2.4\,10^8$
e	22	8	4	1	5	16	$5^{16} \approx 1.5\,10^{11}$

Table 4.3: Characterization of McCallum's mazes with `Active Memorization I`.

and actions were deterministic, and reinforcement was obtained when the agent achieved the goal position (positive) or when the agent collided with a wall (negative). After a positive reinforcement, the agent was placed in a randomly free position of the maze. In our case, we modify the reinforcement function: the agent obtains a positive reinforcement only when it reaches the goal position, and receives a negative reinforcement when it exceeds the maximum number of actions allowed. Note that *no* negative reinforcement is given to the agent when it collides with a wall.

It is important to note that there exist different ambiguous states in each maze that produce perceptual aliasing. These states cannot be avoided (e.g. the ambiguous perception of wall at North and South, but free East and West, in the *b* maze) in order to achieve the goal position. Then, it is not possible to avoid the perceptual states. Also, the best memoryless deterministic policy can only solve a part of the possible problems (e.g. 60% in the *b* maze). Then, it is necessary to disambiguate this states in some way.

We use the CANDID algorithm and the `Active Memorization I` approach to solve these mazes. In this case, a memory bit and an action (`Memorize`) to update this memory, are added to the agent. The execution of the `Memorize` action sets the memory bit on when it is off, and sets it off when it is on.

To test the performance of the new approach, 50 runs were carried out for each maze. Each run consisted in applying the CANDID algorithm and the active memorization approach in order to learn a reliable behavior able to reach the goal position of the maze from any other location.

To learn this behavior the agent performs a sequence of experiences. In each experience, the agent is randomly situated in a non-goal position with the memory bit set to 0. The agent executes actions until it achieves the goal or exceeds the limit of allowed actions. In both cases, a new experience is performed. This cycle stops when the agent has performed 300 consecutive experiences without changing the policy.

The CANDID algorithm has been run with all heuristics activated, because it shows a better performance. The maximum number of actions allowed to be executed in a trial has been set initially to 20 without any search for the best value. Only in cases where the policies obtained could be clearly improved, the number of actions allowed has been reduced (these cases are shown in the table: a and e mazes). The parameter N has been set to 250 without any search for the best value. To ensure the reliability of the learned behavior, each failed experience has been repeated with a limit of 1000 times (this parameter could we lowered); it has not been set to this value after an optimization search.

Results are shown in table 4.4. For each maze and a maximum number of actions allowed (*external and internal actions considered*), the following informations are shown:

1. The *average number of steps* (for 100 randomly chosen initial locations of the maze) that needs the learned behavior in order to achieve the goal position. As in all other items, the mean and median values obtained for 50 runs are shown (in this order). The median value is more adequate to express the performance of the learning algorithm because it indicates the most common performance. These values are measured when the algorithm has finished the learning of the general and reliable behavior. This number does not include the memory actions executed, because they do not perform any step in the arena.

2. The *maximum number of steps* needed to reach the goal. It is the maximum number of steps needed in order to reach the goal state in 100 executions of the learned behavior, each of them starting from a randomly chosen initial location. It expresses the number of steps the agent has to execute to reach the goal position in the worst case. The mean and the median values for 50 runs are shown, as in the previous case. It is interesting to note that usually, the maximum number of steps allowed is a lot higher than the maximum steps of the learned policy. This means that the value for this parameter of the algorithm is not decisive for obtaining an efficient behavior. This number does not include the memory actions executed.

3. The *minimum number of steps* needed to reach the goal. It is the minimum number of steps needed in order to reach the goal state in 100 executions of the learned behavior, each of them starting from a randomly chosen initial location. It express the number of steps the agent has to execute in order to reach the goal position in the best case. As in the previous cases, the mean and the median values for 50 runs are shown (in this order). This number does not include the memory actions executed.

4. The *number of experiences needed to learn the maze*. It shows the number of trials the algorithm has to execute in the maze in order to learn the desired behavior. As in the previous cases, the mean and the median values for 50 runs are shown (in this order). This is a useful index of the speed

of the algorithm in learning the maze. Another usual index in the literature for this concept is *the number of steps performed to learn the maze*. These two indexes cannot be directly compared. In our case, it is not adequate to multiply the maximum number of actions allowed in a trial by the number of trials performed, in order to obtain the number of steps performed, because a trial usually takes less actions than the maximum number of actions allowed and because actions executed does not correspond exactly to steps performed in the arena.

In all cases for each of the 50 runs, the CANDID algorithm and `Active Memorization I` have found an acceptable behavior. The best behavior found, for each of the McCallum's mazes, can be seen in appendix E.

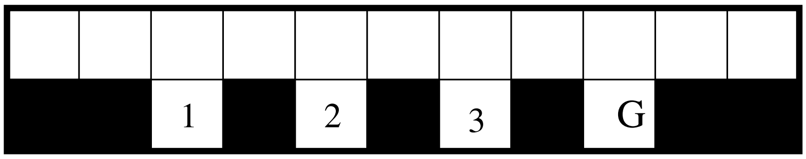
Maze	*Actions allowed*	Average steps	Maximum steps	Minimum steps	Experiences to learn the maze
a	20	(4.6, 4.3)	(8, 8)	(1.2, 1)	(75.8, 42)
a	9	(4.6, 4.6)	(7, 7)	(1.3, 1)	(257.4, 222)
b	20	(5.4, 5.3)	(8.7, 9)	(1.9, 1)	(224, 127)
c	20	(7.1, 7.2)	(11.4, 12)	(2.8, 3)	(203.8, 135)
d	20	(8, 7.9)	(13, 13)	(2.4, 1)	(495.1, 392)
e	20	(7.3, 7.2)	(13.3, 14)	(1, 1)	(204.9, 130)
e	12	(5.1, 5.1)	(8.4, 8)	(1, 1)	(287.1, 250)

Table 4.4: Results for McCallum's mazes with `Active Memorization I`

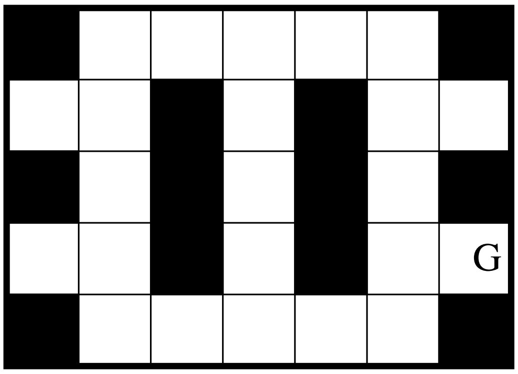
This is a set of simple mazes that has been used to test the different approaches to solve the perceptual aliasing problem.

Our approach can solve all these mazes after a moderate number of experiences performed. The quality of the behaviors obtained is similar to the ones obtained in other approaches, but a lot better than the obtained with the stochastic approach (see [46] and [95]) that, for instance in the *b* maze, needs 8 steps on average to solve the maze.

From the point of view of the number of experiences required to learn the behavior, it seems that our approach learns a lot more quickly than `ZCSM` [46] (*b* maze: 400 trials for 5.3 mean steps behavior x 40 of population per generation = 16000 learning trials) and `UDM` [119] (*b* maze 2500: steps), similar to `Transition Hierarchies` [155] (*b* maze: 500 steps), and slower than `NSM` [123] (*b* maze: 150 steps). The performance that we measure in the number of experiences needed to learn the behavior, does not allow a comparison with the Branch and Bound approach in [95] and the set of algorithms shown in [97], because the way they learn to solve problems is not based on steps or trials.



57-states maze



89-states maze

Figure 4.4: 57-states and 89-states [97]. The numbers inside the cells are perceptible by the agent.

Littman, Kaelbling and Cassandra mazes [97]

The McCallum's mazes are a simplified implementation of a navigation task that a robot must learn. Another kind of navigation problems are proposed in [97]. The two proposed mazes (see figure 4.4.1) are similar to the ones described by McCallum, but they are more complex due to the lack of information from the robot compass. That is, the robot only perceives the presence/absence of a wall immediately in *front, left, right* or *rear*, but it does not know its orientation in the maze. In this case, the robot has 3 actions to move around the maze: `move forward`, `rotate right` and `rotate left`.

If lack of information is the cause of perceptual aliasing, in the 89-states and 57-states mazes, the perceptual aliasing problem is more acute than in the McCallum's mazes. The complexity parameters of these mazes are shown in table 4.5. It can be seen that the relation *real-states* vs. *perceptual-states* (that is, the number of real states represented by a perceptual state) is higher than in the McCallum's mazes.

Maze	*Real states*	*Percep. states*	*Number actions*	Mem. bits	Actions used	States	Number of policies
57-states	57	22	3	1	4	44	$4^{44} \approx 3.1\,10^{26}$
89-states	89	14	3	1	4	28	$4^{28} \approx 7.2\,10^{16}$

Table 4.5: Characterization of 57-states and 89-states mazes for `Active Memorization I`.

As in the previous case, reinforcement is obtained only when the agent reaches the goal position (positive) or when it exceeds the maximum number of actions allowed (negative). Note that *no* negative reinforcement is obtained when the robot collides with a wall.

In order to learn the adequate behavior, the agent performs a sequence of experiences. In each experience, the agent is randomly located in a non-goal position with the memory bit set to 0. The agent executes actions until it achieves the goal or exceeds the limit of allowed actions. In both cases, a new experience is performed. That cycle stops when the agent has performed 500 consecutive experiences without changing the policy.

The CANDID algorithm has been performed with all heuristics activated, because it shows a better performance. The maximum number of actions allowed in a trial has been set initially without any search for the best value. The parameter N has been set to 250 without any search for the best value. To ensure the reliability of the learned behavior, each failed experience has been repeated with a limit of 100 times (but it could we lower or higher; it has not been set to this value after a search for the best value).

To test the performance of CANDID with `Active Memorization I`, 50 runs has been performed for each maze. In all cases an acceptable behavior is found. Results are shown in table 4.6.

Maze	*Actions allowed*	Average steps	Maximum steps	Minimum steps	Experiences to learn the maze
57-states	55	(18.1, 17.8)	(32.4, 34.7)	(1.4, 1.3)	(618.1, 242)
89-states	75	(25.5, 25.2)	(47.4, 47.9)	(1.9, 1.5)	(627, 604)

Table 4.6: Results for Littman, *et al.* mazes with `Active Memorization I`
.

Up to this moment, we are only aware of the results obtained for these mazes with the set of algorithms shown in [97]. In the original version, the aim consisted in finding optimum policies for these mazes. In this paper it is said that the mazes present nosy transitions and observations but, unfortunately, the values for this noise are not shown. We have tested the mazes without noise. *We must remark, then, that the tested environments are not exactly the same.*

The algorithms presented in [97] and our approach are very different. All those algorithms use a world model of the environment and does not measure the time spent to learn a behavior in terms of number of trials or number of steps. Then, the efficiency in learning a behavior is neither comparable.

The only way to compare the two approaches is by measuring the performance of the learned behaviors. In our case, *always* (for 50 runs) a solution has been learned for each maze (this is not the case for the algorithms presented in [97]). The learned behaviors (see appendix E) are acceptable and show a lot better performance than the ones described in [97] (best behavior by using the hybrid algorithm, median number of steps for 57-states maze: 16, 100% of achieving goal; median number of steps for 89-states maze: 40, 57.8% of achieving goal). The improvement can be seen for the 89-states maze. Our approach always solves the problem with 25 steps in median. The other algorithms needed 40 steps in median and only solves 57.8% of problems. *Nevertheless, it is important to note that noise has not been considered in our experiences, and the comparison has to take into account this fact.*

It was surprising to detect that those mazes could be solved without the aid of memory actions. Although the perceptual aliasing is more marked in these mazes than in the McCallum mazes, the set of actions of the agent allows to find general policies that, within the limited number of actions, can solve the problem from any initial location. In the case of the 89-states maze, policies that use memory actions are more efficient than memoryless policies, and the 50 runs of the algorithm returned only the former kind of policies. But in the case of the 57-states maze, surprisingly, there are memoryless policies more efficient than policies that use memory actions to solve the task. In this case, some of the 50 runs obtained memoryless policies, while the others obtained policies that effectively use memory actions. Thus, table 4.6 shows for this maze mixed results of these two kinds of policies. In table 4.7, statistics for the two kind of policies are shown.

Maze	*Actions allowed*	*Uses Memory?*	Average steps	Maximum steps	Minimum steps
57-states	55	Yes	(18.4, 19.7)	(40.3, 40)	(1.2, 1)
57-states	55	No	(14.6, 14.4)	(28, 24)	(1.4, 1)

Table 4.7: Comparison of polices for 57-states maze.

In this case, the least efficient policies could be avoided by reducing the maximum number of actions allowed to around 30 (that, in this case, implies a maximum number of steps about 35). This would eliminate the first kind of policies shown in the table (that in the worst case need about 40 steps) and would improve the average results.

The learned behaviors shown in appendix E are the best found for this task. In the case of the 89-states maze, the behavior uses the memory action, but in the case of the 57-states maze, the behavior does not use it[7].

4.4.2 Limited perception II

The tasks of this section are more difficult to solve than the task of the previous one. This is due to the *extreme lack of perceptual information* that the agent has to deal with. In all these cases, for each perceptual state the use of more than one action is needed. So, these tasks will be useful in order to test the second kind of active memorization architecture described in section 4.3.2: `Active Memorization II`.

Cassandra, Kaelbling and Littman's maze [33]

This is a simple maze designed by Cassandra, Kaelbling and Littman [33] to show the performance of the `Witness` algorithm. It been also used by Parr and Russell [145] for testing the `SPOVA` algorithm. This maze consists of a 4x4 grid that is shown in figure 4.5. The feature that makes difficult this maze is that the agent is entirely blind, that is, it cannot perceive walls nor anything. As in other cases, the agent is placed randomly in the grid, and the goal consists in achieving the "G" position. Reinforcement is given to the agent only when it reaches the goal position (positive) or when it exceeds the maximum number of actions (negative). Note that *no* negative reinforcement is given to the agent when it collides with a wall. In this case, the agent remains in the same position. In the original version of the maze, only positive reinforcement, when the agent achieved the goal state, was given to the agent. For this maze, the actions available to the agent are `Move-North`, `Move-South`, `Move-East` and `Move-West`.

[7] In the behavior for the 57-states maze, the memory action is defined to be executed for some states, but it is not really needed. We can note this because in each situation, the action to be executed for the two different memory values is the same, or one of them is nil, or one of them is not an external action (one of them is the memorize action). Remember that *two memory is needed because in some situations, different external actions have to be executed depending on the real state*. Then, memory is used to disambiguate the current state. But in this behavior, there are no situations that execute two different external actions depending on the memory bit. On the contrary, the behavior shown in appendix E for the 89-states maze uses memory because we can found some situations that execute different external actions depending on the memory bit

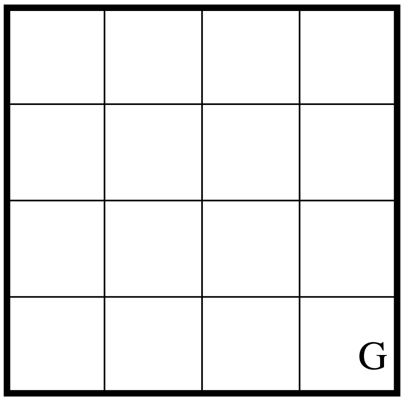


Figure 4.5: 4x4 maze [33].

This maze becomes a difficult task to solve due to the absence of perceptual information. There are 16 different states, all of them aliased in 1 perceptual state! This means that after the execution of any action by the agent (for instance a), its perception does not change (because it is always the only one possible). When the agent selects, depending on the perceptual state, the next action to be executed, it will have to choose the same action (a) as in the last step, because the perceptual state remains the same; and so on. In this case it is appropriate to use the `Active Memorization II` architecture, because *in order to overcome the always-choosing-the-same-action impasse, an oscillator that changes its value automatically after the execution of any action, becomes necessary* (see section 4.3.2). In this case, we use one memory bit and a binary oscillator. The complexity parameters for this maze are shown in table 4.8.

Maze	*Real states*	*Percep. states*	*Number actions*	Memory bits	Actions used	States	Number of policies
4x4-grid	16	1	4	1 + 1 osc.	5	4	$5^4 = 625$

Table 4.8: Characterization of the 4x4 grid with `Active Memorization II`.

To learn the adequate behavior, the agent performs a sequence of experiences. In each experience, the agent is randomly situated in a non-goal position with the memory bit and the oscillator set to 0. The agent executes actions until it achieves the goal or it exceeds the limit of allowed actions. In both cases, a new experience is performed. This cycle stops when the agent has performed 200 consecutive experiences without changing the policy.

CANDID has been run for this maze with all heuristics activated, as in the previous cases, because it shows a better performance. The maximum number of actions allowed has been set without searching for the best value. The parameter N has been set to 100 without any search for the best value. To ensure the reliability of the learned behavior, each failed experience has been repeated with a limit of 200 times, but it could we lower or higher. No optimization search has been done for this value.

To test the performance of `Active Memorization II`, 50 runs have been performed for the maze. Results are shown in table 4.9. In all runs, the optimal behavior was found (see appendix E).

Maze	*Actions allowed*	Average steps	Maximum steps	Minimum steps	Experiences to learn the maze
4x4 Grid	20	(4.83, 4.88)	(6.64, 7)	(1.07, 1)	(7.78, 9)

Table 4.9: Results for the 4x4 Grid

We are only aware of results for this maze by using the `Witness` and `SPOVA` algorithms. These algorithms use a model of the world that is not necessary in our approach. Behaviors obtained are comparable with the ones obtained those algorithms, but they were obtained much more quickly with our approach: with an average of 8 or 9 experiences the agent learned the best policy.

Complex Mazes with simple solutions

Another interesting problem is the maze presented by Wiering and Schmidhuber in [208] (see figure 4.4.2). It is a big maze of 952 states. Following the authors, this is a moderately complex maze for `Q-learning` with complete information (for instance, the current position in the maze), but a maze impossible to learn for `Q-learning` with incomplete information (concretely, as in the McCallum's mazes, with local information about presence or absence of a wall in the surrounding cells).

Nevertheless, the maze presents a very simple solution that consists in going forward while counting the number of open doors at the left, and to turn left when the fifth open door at the left appears. In [208], this solution is found by searching in the space of simple programs (see the automatic programming approach in section 4.2.5) that can be built from a simple set of instructions and conditionals (that are described in the paper).

We have tested CANDID using the `Active Memorization II` approach in this maze, with the set actions and perceptions proposed by Wiering and Schmidhuber

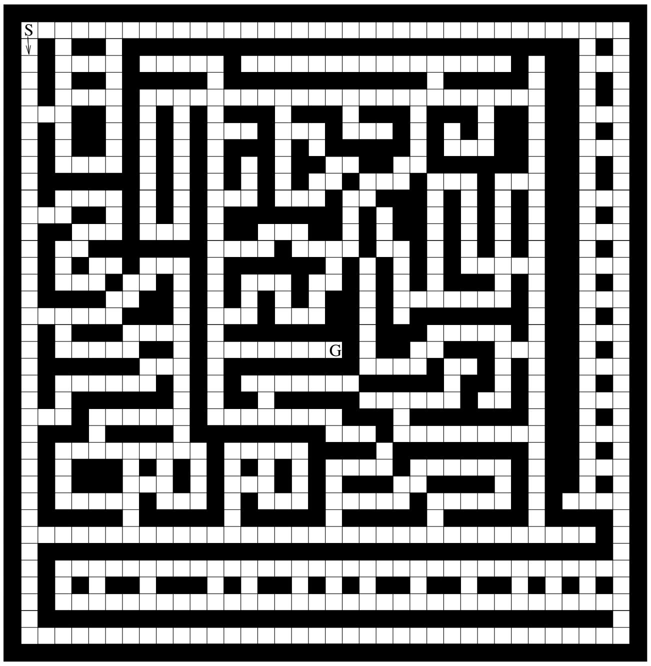


Figure 4.6: 38x39 Grid [208].

in the paper, except the `Jump` action that was replaced by a memorization action. The set of 9 actions used is:

1. `Move forward until front blocked, then rotate left`
2. `Move forward until front blocked, then rotate right`
3. `Move forward until left field is free, then rotate left`
4. `Move forward until left field is free, then rotate right`
5. `Move forward until left field is free, then do not rotate`
6. `Move forward until right field is free, then rotate left`
7. `Move forward until right field is free, then rotate right`
8. `Move forward until right field is free, then do not rotate`
9. `Memorize`

The memory register used is of module 5, similar to the *nr-times* parameter used by the authors in [208]. When the memory action is executed, it is added 1 module 5 to this memory register.

As it can be seen, the perceptions are included in the actions (that is, one action is performed until one perception is achieved). Perceptions are, then, taken into account and they do not need to be represented again separated from the actions. In this case, the agent has defined only one perceptual state. As in the previous maze, after executing any external action, the agent still perceives the same perceptual state. This implies that the agent would choose (with the standard architecture) the same action all times. Similarly to the 4x4 Grid case, we solve this impasse with `Active Memorization II` by adding a binary oscillator to the agent.

In order to learn the adequate behavior, the agent performs a sequence of experiences. At each experience, the agent is placed in the same initial state (up-left corner of the maze, with the south orientation), with the memory register and the oscillator set to 0. The agent executes actions until it achieves the goal location (in the center of the maze), until it bumps against an obstacle, or until it exceeds the maximum number of allowed actions. In these cases, a new experience is performed. In the first case, positive reinforcement is returned. In the second and third cases, negative reinforcement is returned. This cycle stops when the agent has performed 100 consecutive experiences without changing the policy.

The CANDID algorithm has been performed with all heuristics activated, as in the previous cases. The maximum number of actions allowed in a trial has been set to 70 without performing any search for the best value. The parameter N has been set to 250 also without any search for the best value. In this case, to ensure the reliability of the learned behavior with a repetition of failed experiences is not necessary, because in this maze there only exists one setting to solve (in all cases the robot starts in the same initial location in the same maze). In order to test the performance of `Active Memorization II`, 50 runs have been performed. In all runs, the optimal behavior was found (see description of the obtained behavior in appendix E).

Results are quite interesting. In [208], `Q-learning` with complete information, `Q-learning` with incomplete information, `Q-learning` with incomplete information and a window of the last 5 actions executed, and the `Levin Search` algorithm, were tested. As in our experiment, the agent is rewarded when it achieves the goal location and it is punished when it collides with any obstacle. In both cases, the agent is reset to the initial location and a new experience is started. Unlike our experiment, the agent is neither reseted nor punished when a maximum number of actions have been executed.

On one hand, `Q-learning` with complete information, that is, without the perceptual aliasing problem, solved the maze in $7 \cdot 10^5$ steps. On the other hand, no any of the other two variants of `Q-learning` solved the problem. The `Levin`

`Search` solved the maze but it needed more than $9.7 \cdot 10^7$ steps. *In contrast to all this,* CANDID *with* `Active Memorization II` *obtained the optimal behavior in* 7031 *trials average,* 4240 *trials median. If we measure these results in the number of steps performed in the arena, our approach solved the maze in* $1.2 \cdot 10^6$ *average,* $7.6 \cdot 10^5$ *median of steps performed. The most expensive run needed* $3.0 \cdot 10^6$ *steps and the less expensive needed 76064 steps. We can conclude from this results, that ours is the best approach tested for this maze*[8].

It is interesting to note that *our approach showed even a better performance than* `Q-learning` *with complete information.* The explanation for this good performance is that incomplete perception produces a profitable generalization that reduces the size of the policy table and that, in this case, speeds up the learning process.

It is also interesting to note that the `Q-learning` variant that used a window for remembering the last 5 actions executed, although *in theory* could solve the problem, failed in practice to solve it. The cause is that the relevant information for solving the problem consists only in the fifth last past action, becoming noise the other ones (from first to fourth last actions). This noise slows the performance of the algorithm. *This is the reason why our more selective approach* `Active Memorization II` *solved the problem while the window approach failed.* This was predicted in section 4.2.8.

4.5 Conclusions

In this chapter, a new strategy to solve the perceptual aliasing problem, that is suited to embedded agents, has been proposed.

This new proposal has been made after a critical study of other approaches developed to solve the same problem. From this study, we have proposed a new strategy, implemented through two different architectures, `Active Memorization I` and `Active Memorization II` (see section 4.3.2), that must be used depending on the task the agent is facing.

The main difference between the active memorization approach and other memory approaches consists in the different *selective procedure* used to store the relevant information that allows to solve the perceptual aliasing problem. Our approach is much more selective these approaches, for instance the window approach

[8]One of the main causes for the good performance of CANDID in this task is that all heuristics are activated. As this task consists in solving only one possible setting (there is only one initial location in the maze), when the agent receives the first positive reinforcement (solves on problem), we can state that it has learned to solve the entire task (solves all the problems). This is the best possible case for CANDID and the set of heuristics, because it has been proved (see appendix C) that in those conditions, the learning algorithm explores the space of possible policies without repeating any one of them. This non-redundant exploratory behavior is the main responsible for the efficiency in learning the behavior.

(see section 4.2.8) in which all last t actions and/or situations experienced by the agent are automatically stored. These memory approaches (see section 4.2.8) become passive in the way that historical information is stored, because it is done automatically. In this way, relevant and not relevant information is stored and taken into account in order to solve the perceptual aliasing problem. When the amount of irrelevant information stored becomes too high, the learning of the task becomes too difficult.

On the contrary, our approach does not assume the continuum hypothesis (that all last t situations are relevant for solving the perceptual aliasing problem). This implies that in our approach, the agent must learns not only how to solve the problem with external actions, but also how to actively collect the necessary information to disambiguate aliased states. For instance, when the agent passes through an state and it would be useful (in order to solve the perceptual aliasing problem) to remember this, the agent must actively memorize it by executing a memory action.

Our approach is also different to previous attempts to solve the perceptual aliasing problem by using memory actions (for instance Littman's and ZCS approaches; see section 4.3.1) because our approach does not show the exponential growth of actions that makes these previous attempts computationally intractable.

Also on the merits side, it is interesting to note that our approach can solve the perceptual aliasing problem avoiding some constraints that are present in other methods that make no use of memory. For instance, it is possible to visit aliased states without any risk; this was not possible in the *ignoring* and the *avoiding aliased states* approaches (see sections 4.2.1 and 4.2.2). It is possible to learn a behavior without having a model of the environment, which is suited to the view of an embedded agent that learns only interacting with the environment without any *a priori* knowledge (in the *belief vectors* approach, a world model is necessary; see section 4.2.3). Finally, in our approach it is not necessary to learn a world model of the environment; in the *disambiguate states by learning a model* approach (see section 4.2.3), this is necessary.

We must remark that our approach, by being independent from a world model, allows learning general behaviors. Our approach is, therefore, an adequate extension to the CANDID algorithm.

Experiments show that the *active memorization* approach is useful to solve the perceptual aliasing problem. It works well in a extensive set of navigation environments and solves problems that include an extreme lack of perceptual information and tasks that include a large number of real states (see results in section 4.4). It solved comparatively equal or better than other approaches the proposed experiences.

Surprisingly, in some cases, the approach produced even better results than algorithms that use a complete information of the environment. Specifically, it must be noted that in the last maze, the new approach shows a better performance compared the `Q-learning` algorithm *with complete information.*

It is interesting to note that some of the obtained policies for those navigation tasks, describe a behavior similar to the ones reported in the literature as *landmark navigation.* That is, the agent learns to recognize and remember some states that show unique features. Those landmarks are non-ambiguous states that are used to eliminate the perceptual aliasing problem.

In spite of this, the active memorization approach shows some problems and open questions that could be interesting to study.

One of them is about the number of memory registers that are needed to solve a given problem. Can it be determined *a priori*? If this is not feasible, an alternative question would be if it is possible to design a procedure that could determine (while the learning algorithm is running) when the used memory bits are insufficient to solve the perceptual aliasing problem. This procedure would be used to dynamically increase, accordingly, the memory resources of the agent.

Another open question is the following: we have used the active memorization approach with the Candid algorithm, but can it be extended successfully to other algorithms (as `Q-learning`)? The Candid algorithm is not designed to find optimal policies, but could the active memorization approach be applied to optimization algorithms in order to find them?

Chapter 5

Scaling up: Incrementality and Development

5.1 Introduction

As it has been remarked in chapter 2, in order to solve complex tasks it is important to deal with the perceptual aliasing problem (because it usually occurs in real applications and in generalization processes), but *it is also very important to tackle with the difficulties that appear when a large chain of actions is needed to solve the task* (i.e., scale up to problems with a long delayed reinforcement).

The main difficulty of this kind of tasks consists in that reinforcement is very rarely obtained. We must keep on mind that the reinforcement signal is the only information that guides the learning and, in this case, the amount of guiding information available for an efficient learning becomes insufficient. See section 2.2.1 for a description of the difficulties that appear when dealing with a long delayed reinforcement.

There are several ways of facing this problem. The discriminating feature of them is the properties they consider to be the cause of the delayed reinforcement. The usual approaches to solve the problem are: to resort to a teacher, to exploit *a priori* domain knowledge, to use a smart exploration technique, to learn in a incremental way, and to build-up a hierarchy of intermediate actions. All these approaches will be analyzed in depth in the following sections. Although we will discuss all the approaches, we will advocate for the last one, which we consider more effective for our purposes and more philosophically appropriate for embedded agents than the others.

5.1.1 Teaching

The first approach we will discuss is the one that solves the delayed reinforcement problem by teaching the agent. Following this approach, the source of the delayed reinforcement problem is due to a bad definition of the task. If it is hard (or impossible) to learn from a long delayed reinforcement, then reinforcement has to be more frequent. This obvious approach consists in reducing the delay in the response of the teacher (that is, it consist in redefining the task to give more frequently a reinforcement signal to the agent). This can be done in several ways.

One possible way could be by giving *intermediate reinforcement signals* to the agent in addition to the reinforcement that indicates the final achieving of the goal. One example the use of this technique is found in Matarić's work [117]. Another one is Caironi and Dorigo's work [32]. In these cases, a complex task can be eased by giving a more frequent reinforcement signal to the agent.

A second way to solve the delayed reinforcement problem by teaching consists in giving to the agent what it is called an *immediate reinforcement*. It consists in returning, for each action performed by the agent a local reinforcement signal that indicates whether the performed action is the adequate one or not. This approach is used, for instance, in the first phase of the methodology proposed by Dorigo and Colombetti in [61]. It is also used in some other of their works ([60], [57], [48] and [49]).

Finally, a more extreme approach consists in guiding the learning process of the agent , when it is not progressing well, by proposing for some perceptions the action to be executed. In this case, the agent assumes the proposed action as if it were the one chosen by itself and modifies its values and policy tables accordingly. An example of this *supervisory* (or *example based*) approach is, for instance the one presented in [47] for Markovian tasks.

In order to unify all the approaches mentioned, they can be situated in a imaginary line with two extremes that represent the not-so-much delayed reinforcement approach on one end and the supervised learning on the other.

Generally, we consider that the immediate reinforcement and the example based approaches are not appropriate for embedded agents.

In the case of immediate reinforcement, it is problematic to determine when the executed action is the adequate one. It has been indeed observed by Dorigo and Colombetti [48] the difficulty of designing good teacher programs that return an immediate reinforcement to the agent (see section 2.1).

One naive approach to the design of those kind of reinforcement programs could consist in detecting whether the execution of the action draws the agent nearer to the goal state. Unfortunately, a function that indicates the proximity of the agent to the goal is not always adequate to deliver a reinforcement signal. This kind of functions can drive the agent to local minima. An action that draws the agent nearer to the goal does not always imply that it is the right action to execute. Thus, the design of an adequate reinforcement function is not easy. Nevertheless, an interesting approach to circumvent this problem could consist in developing learning mechanisms that can operate with not totally trustworthy reinforcement functions [32].

Another difficulty for developing good functions that return immediate reinforcement is the difficultness of putting ourselves in the place of an embedded agent, considering fully and completely its sensory system and its dynamics. Something that can be reasonable for us given our sensory and motor system, can be not reasonable for such an agent. Researchers in robotics know about this problem that also appears while programming real world robots.

Moreover, in case that the teacher could solve this problem and put herself in the "skin" of the embedded agent, in order to design a program that returns a trustworthy immediate reinforcement, she must knew completely the desired behavior. In this case, it would be better to directly program into the agent the desired behavior than making it learning the behavior.

In the case of the example-based approach, all the previous criticisms could also be applied, with the possible exception of the need of knowing *completely* the behavior. In this case it is only needed to know some examples of actions to execute in some circumstances, that partially describe the behavior.

For the case of using intermediate reinforcement signals, we think that it could be useful, but that it also could present, in a less extent, some problems similar to the described above. Nevertheless, we think that the lighter approach of Matarić [117] and Caironi and Dorigo [32] could be useful in some cases.

In spite of that we do not think that the application of teaching mechanisms is the best approach to solve the problem of a long delayed reinforcement for embedded agents, the application of these techniques to the CANDID algorithm is still feasible. Unfortunately, the use of active memorization complicates the scenario: the immediate reinforcements or the examples have to consider also the memorization actions (not only the motor actions).

5.1.2 Partial modeling

Partial modeling consists in giving an *a priori* set of clues about the right behavior of the agent, in order to make easier the process of finding the goal state by avoiding the exploration of useless or dangerous actions.

This can be done by giving a set of reactions to a set of stimulus so as to constrain the size of the space of policies. In this case, the agent has an *a priori* knowledge that speeds up its learning. The approach can be described as "*do not make the agent learn what it is known; it is better to program it*".

For agents that build models of the environment in order to learn, this approach could consist in giving to the agent an incomplete or partial model of the environment.

We think that this approach can be useful in some cases, but that using *a priori* knowledge poses some of the problems described in the previous approach. On the other hand, in order to learn general tasks it is not appropriate to use learning algorithms that build models, because they are useful only for one instance of the task, not for all problems of the task. For instance, building a model of a room could be useful to learn the specific behavior *"go out of this room"*, but it is not useful to learn the general behavior *"go out of any room"*.

5.1.3 Guided exploration

In some domains, a positive reinforcement signal can only be obtained in very few states that are also very far away from the initial states. Under these conditions, finding by *trial-and-error* the goal states that provide a positive signal becomes very hard. Learning becomes, then, very hard because whatever the agent does, it usually obtains the same negative reinforcement. In this case, most algorithms usually tend to fall into the repetition of unsuccessful policies that were tested in previous experiences. The repetition of unsuccessful policies, obviously implies an impoverishment of the agent's learning performance. Some complexity studies about the tendency of repeating the exploration of unsuccessful states for some classical reinforcement learning algorithms in Markovian domains have been made in [195] and [194].

In order to alleviate this problem, smart exploration techniques of the state space that avoid redundancies and visit the less known parts of the state space while searching for the goal states, have been designed. This approach has been called *guided exploration* and it is focused on finding, as quickly as possible, a first positive reinforcement. Once the first positive reinforcement signal is obtained, the problem becomes easier because, from the trace of the successful behavior, a guide to find again a new positive reinforcement can be extracted. Different techniques have been described for the implementation of this approach; see [195] and [194] for an exhaustive review.

Unfortunately, those techniques are useful only when a clear differentiation among the real states of the environment can be made. This invalidates the use of these techniques for embedded agents with limited perception, except for those approaches that solve the perceptual aliasing problem by building models of the environment (see for instance [128]).

Nevertheless, the CANDID algorithm does not need to use these techniques. It can be proved (see appendix C) that the use the CANDID algorithm together with the whole set of heuristics proposed in the chapter 3 *does not redundantly repeat any policy until a first positive reinforcement is reached*. The original CANDID algorithm (without any heuristic) also shows this property under the additional constraint of not passing through the same perceptual state more than one time per experience.

5.1.4 Incremental learning

Incremental learning is a powerful idea that allows to learn complex tasks in a natural way for embedded agents. It is based on the piagetian idea of "it can only be learned what it is almost known" [20]. That means that an agent can only learn to do things that are very close to its current abilities. Following this idea, the only way to learn complex tasks consists in learning in a incremental way more elaborated abilities until the agent can afford the original complex task.

There are several approaches that can be tagged as "incremental", although they do not explicitly adhere to the piagetian statement. We divide these approaches into two different perspectives, that will be thoroughly analyzed in the following sections:

1. **Incrementality on the size of the task:** It consists in the incremental learning of more complex problems of the same kind of task. It implies an incremental *growing of the performance of the same behavior*. In the early stages, the easy problems of the task are presented. Once they are learned, more complex problems are proposed. This progression continues until the agent has learned to solve the more complex problems of the task with a single behavior. This approach has not been extensively studied, except by Ring [155], who calls it *continual learning*. It will be described with more detail in the next section. In the same section, its application to the CANDID algorithm will be tested.

2. **Incrementality on the kind of task:** Also called *hierarchical learning* [92] [82], *behavior coordination* [9] [74] and *hierarchical development* [112]. It consists in learning to solve a task by starting from other previously learned behaviors. These intermediate behaviors are more suited to learn the complex task than the agent's primitive set of actions. It implies the incremental *growing of new behaviors* able to deal with more complex tasks. This approach will be further analyzed in section 5.3.

Incremental approach seems more suitable for embedded agents than other approaches mentioned above. This approach does not need a deep knowledge of the dynamics of the agent interaction with the environment (that was needed in the teaching and in the partial modeling approaches). It only requires to know the intermediate problems to learn or the intermediate behaviors to learn for the learning of the complex task. This is usually more easier to state than the adequate *a priori* knowledge or the appropriate immediate reinforcement function.

On the other hand, although a teacher is necessary to indicate the sequence of problems or behaviors to be learned in order to solve the complex task, this does not imply a direct intervention on the way the agent uses it sensors or chooses the actions to execute, thus avoiding the problems it could convey for embedded agents.

5.2 Incrementality on the size of the task

The first incremental approach to achieve complex skills, implies the control of the order of experiences performed by the agent. Following the idea of "learning first simple things to learn more complex tasks later", the agent will learn first to solve simple problems (smaller) in order to learn more complex problems (bigger) later. In this case, the behavior incrementally learned is always to solve the same task. An instance of this approach is the CHILD architecture, proposed in [155].

One example of a prototypical application of this approach is shown in figure 5.1 (taken from [155]). This figure presents a sequence of nine navigation tasks, sorted by its complexity form the simplest to the most complex one. The agent solves the navigation tasks when it achieves the goal state. Perceptions and actions are the same that were used in the McCallum's mazes (see section 4.4.1). Thus, the agent has to deal with ambiguous perceptions.

All mazes have a similar structure with the goal state located at the end of a corridor, but each maze presents more states, presents more ambiguous states and/or needs more steps to solve the problem than the preceding maze. In this case, it is reasonable to assume that the knowledge of how to solve, for instance, the fourth maze could be useful for learning to solve the fifth one. That is, it would be better to use that knowledge to solve the fifth maze than learning to solve it from scratch.

A set of experiences has been performed in order to test this approach with the CANDID learning algorithm and the active memorization approach. The procedure used to test this incremental approach is defined as follows.

$i := 1$

Execute CANDID to learn the first maze;

Save the learned policy and statistics as the first ones;

while $i < 9$ **do**

  $i := i + 1$

  Execute CANDID for the i-nth maze starting from the $(i-1)$-nth policy and statistics;

  Save the learned policy and statistics as the i-nth ones;

end while

This procedure was repeated for 30 different runs, with different random seed values. The maximal number of actions to be executed in a trial for each maze has

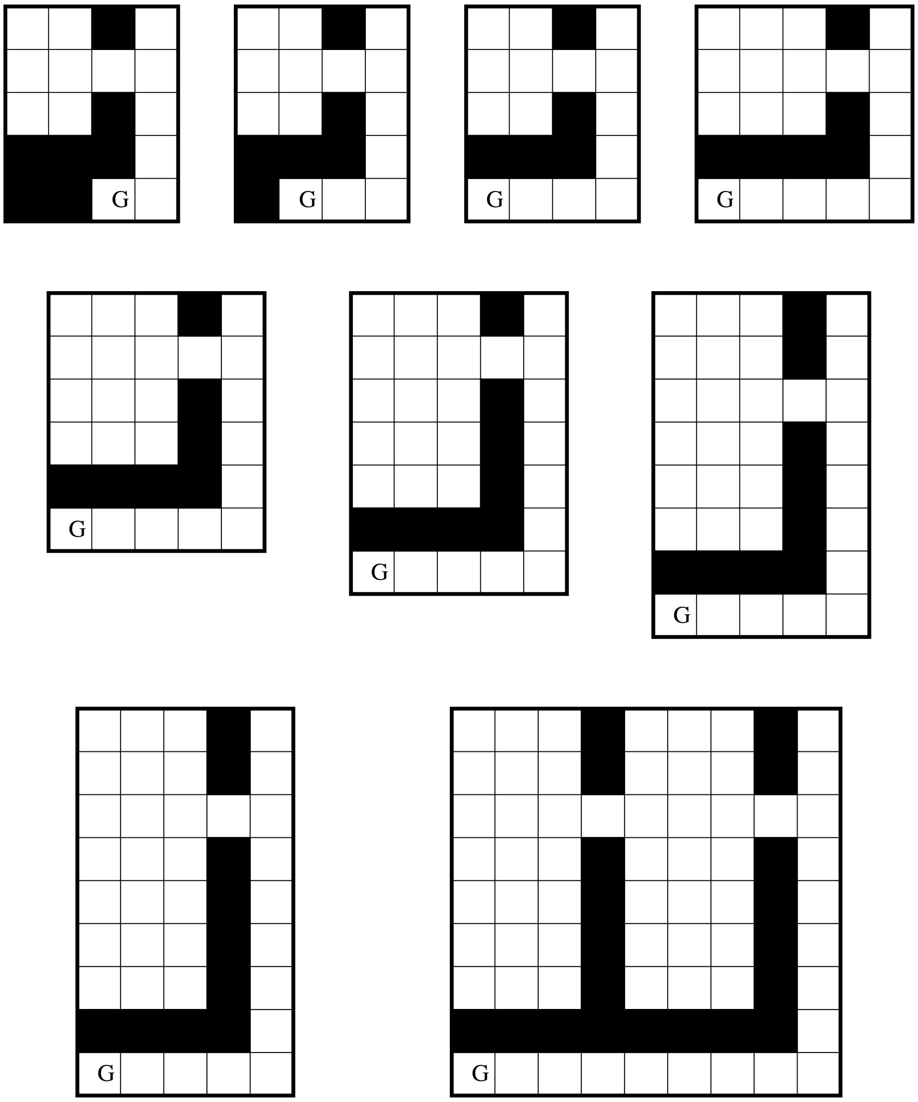


Figure 5.1: Incremental problem [155].

been set without the searching for an optimum performance. It has been increased as the size of the maze increases. In all executions of CANDID, the N parameter was set to 100 and the maximum number of repetitions for a failed experience was set to 100. Not any of those parameters has been obtained after a search for the best values. Results are shown in tables 5.1 and 5.2. Table 5.1 shows the obtained results using the CANDID algorithm together with `Active Memorization I` (with one memory bit) from scratch. Table 5.2 shows the obtained results with the same mechanisms but using the incremental approach.

Tables 5.1 and 5.2 show, for each maze and a maximum number of actions allowed per experience, the following information:

- *Average steps:* After learning the 100% reliable behavior, the average (for 100 randomly chosen problems) number of motor actions (steps) needed to solve a problem. The average and median results for 30 different runs are shown (in this order).
- *Maximum steps:* After learning the 100% reliable behavior, the maximum number of motor steps needed in 100 randomly chosen problems, to achieve the goal state. The average and median results for 30 different runs are shown (in this order).
- *Minimum steps:* After learning the 100% reliable behavior, the minimum number of motor steps needed in 100 randomly chosen problems, to achieve the goal state. The average and median results for 30 different runs are shown (in this order).
- *Experiences to learn the maze:* The number of experiences needed in order to obtain a 100% reliable behavior. In the case of incremental learning (table 5.2), the values exhibited represent the number of experiences needed to learn the reliable behavior for the current maze *starting from the behavior learned for the previous maze.* The average and median results for 30 different runs are shown (in this order).
- *Accumulated experiences:* Only for the incremental results (table 5.2). The total number of experiences needed to learn the current maze in an incremental way *starting from the first maze.* This value is obtained by adding the number of experiences to learn the current maze and the accumulated number of experiences for the previous maze. The average and median results for 30 different runs are shown (in this order).

From the comparison of the two tables, we can note the following:

1. The first maze presents the same results in the two cases (incremental and non incremental), which is obvious because both experiments are in fact the same one.

Maze	*Actions allowed*	Average steps	Maximum steps	Minimum steps	Experiences to learn the maze
First	12	[5.84, 5.84]	[9.2 ,9]	[1.1, 1]	[550, 554]
Second	15	[6.87, 6.78]	[11.2, 12]	[1.6, 1]	[693, 707]
Third	16	[7.54, 7.44]	[11.9, 12]	[2.5, 3]	[1096, 1164]
Fourth	20	[9.40, 9.46]	[15.0, 16]	[2.7, 4]	[603, 558]
Fifth	24	[11.69, 11.62]	[19.5, 20]	[2.1, 1]	[1722, 844]
Sixth	28	[13.39, 13.41]	[22.5, 23]	[1.9, 1]	[1291, 1014]
Seventh	32	[14.71, 14.41]	[25.5, 25]	[2.6, 1]	[931, 604]
Eighth	34	[16.07, 15.81]	[27.4, 28]	[2.1, 1]	[1465, 637]
Ninth	44	[22.19, 21.76]	[35.6, 36]	[4.9, 1]	[1021, 701]

Table 5.1: Results for Ring mazes with `Active Memorization I`

2. In the second maze an aliased state, that was not present in the first maze, appears. It is *necessary* to execute two different actions (`MOVE-EAST` and `MOVE-WEST`) for this aliased perceptual state in order to achieve the goal state. Thus, in contrast to the previous maze, the use of actions to memorize becomes necessary. This explains why the second maze needs such a long number of experiences to learn in an incremental way (although not so much than the first maze from scratch, which indicates that *there really exists a transfer of skills between the two mazes*). The number of accumulated experiences and the number of experiences from scratch are more or less the same.

3. From the second to the third maze in the incremental way of learning, it is not necessary to learn nothing (0 experiences) to obtain a reliable behavior. In fact, the second and third mazes are not different in the number of perceptual states and, although there is one new real state in the third maze, it does not require an action different to the one used in the other real states with the same perception, for a reliable behavior.

4. Only two experiences, in average, are necessary to perform from the third maze in order to learn a reliable behavior for the fourth maze. *This shows that transfer of skills between mazes works well in our approach.* In the fourth maze, three new aliased states appears and, consequently, some experimentation is needed in order to find a reliable behavior for the new maze. *Surprisingly, in this case, the number of accumulated experiences needed for learning the new maze is higher than the one needed for learning from scratch!* At first glance, this could seem disappointing for the incremental approach. A more careful analysis shows that the *quality* of the obtained behaviors is quite different and, consequently, the results are not

Maze	*Actions allowed*	Average steps	Maximum steps	Minimum steps	Exps. to learn maze	Accumulated experiences
First	12	[5.84, 5.84]	[9.2, 9]	[1.1, 1]	[550, 554]	[550, 554]
Second	15	[6.44, 6.33]	[10.3, 10]	[1.5, 1]	[222, 149]	[772, 703]
Third	16	[7.15, 7.08]	[11.3, 11]	[1.9, 1]	[0, 0]	[772, 703]
Fourth	20	[8.89, 8.83]	[13.6, 13]	[2.7, 4]	[2, 2]	[774, 705]
Fifth	24	[8.86, 8.89]	[13.6, 13]	[2.3, 1]	[0, 0]	[774, 705]
Sixth	28	[13.10, 13.11]	[22.7, 23]	[2.3, 1]	[672, 452]	[1446, 1157]
Seventh	32	[13.87, 13.78]	[25.1, 25]	[2.3, 1]	[157, 1]	[1603, 1158]
Eighth	34	[15.42, 15.10]	[28.1, 28]	[2.3, 1]	[5, 0]	[1608, 1158]
Ninth	44	[21.94, 21.07]	[35.8, 36]	[4.0, 1]	[6, 0]	[1614, 1158]

Table 5.2: *Incremental results* for Ring mazes with `Active Memorization I`

comparable. In the incremental case, the average number of steps needed to solve a problem is significantly lower than the needed from scratch. To obtain from scratch a behavior with the same quality that the one obtained in the incremental way, more experiences would be necessary to perform.

5. The fifth maze shows the main benefits of using the incremental approach. Not only the quality of the learned behavior is better than in the non-incremental case, but also the accumulated number of experiences is lower than the number of experiences needed from scratch.

6. For the sixth maze, the same remarks as in the point 4 can be made about the accumulated number of experiences and the quality of the learned behavior. It is also interesting to note that the sixth maze is very difficult to learn from the fifth one. The reason for this behavior is not clear, but it could be related to the kind of memory management used for the fifth maze that becomes incompatible and has to be radically changed for solving the sixth maze.

7. For the seventh, eighth and ninth mazes, the same remarks of point 4 can be made (except maybe for the ninth maze, where the obtained policies in the two cases present a similar efficiency). In the case of the seventh maze, where the average and median number of experiences needed to learn the maze in the incremental way are so different, a particular run of the 30 performed was very expensive (3840 experiences) to find a reliable behavior. The reasons for this bad performance in one run are not clear, but it can be due to a bad luck in the sequence of the randomly generated problems presented to the agent, that could become misleading.

The obtained results show that an effective transfer of skills is possible between mazes. In some cases, only a slight training is necessary in order to increment the abilities of the agent, as can be noted in the third, fourth, fifth, seventh, eighth and ninth mazes. *We think that the main cause for the success of this approach using* CANDID *is that the algorithm is designed for learning general behaviors with incomplete perception.* This feature of the algorithm seems that eases the transfer of skills for similar mazes.

5.3 Incrementality on the kind of the task

There is another approach to incremental learning that is not designed to increase the abilities of the behavior respect to the size of the task, but to increase them with new behaviors that solve more complex tasks. It consists in learning to solve a new task by starting from some previously learned behaviors, instead of using the primitive set of actions of the agent. Those learned intermediate behaviors are more suited for the process of learning to solve the complex task than the primitive set of actions of the agent.

This approach is sometimes called *hierarchical learning* [92] [82] because a hierarchy of behaviors, that increases the abilities of the agent, is learned. It is also called *behavior coordination* [9] [74] because the new behavior that solves the complex task, uses previously learned behaviors and, thus, it can be seen as a coordinator or manager of these previous learned behaviors.

5.3.1 Previous work

The use of a hierarchy of learned behaviors to solve a complex problem has been proposed in several works. We divide these works into two main streams: those where hierarchies are obtained by dividing the state space of the complex task, and those where hierarchies are obtained by decomposing the complex task into sub-tasks. These two approaches are analyzed in the current section.

State space hierarchy

In this approach, it is assumed that there are several levels of representation of the state space of the task. Each level describes the state space with a different grain of detail. These levels are organized in a hierarchical way: the more detailed description, the lower the level. In this case, a set of states on one level are coarsely represented as a single state in the immediately upper level. We will say that the set of states that are coarsely represented in a single state of the higher level, "belong" to it.

One instance of this approach is the `Feudal Q-learning` algorithm [51]. In this work, the goal state is represented in each level. It is recursively defined for level i as the state where the goal state of the lower level *i-1* belongs to, except for the lowest level (the level with the most detailed description) where the goal state is directly defined by the task. For each state in any level (except in the lowest one) a manager is defined that commands over all the managers of the subspace formed by the set of states that belong to it. In this case, the learning procedure can be informally described as follows. Assume that the manager of state s in level i proposes an action a to be executed (this action is considered as the action proposed for the state s in level i). If the action is achieved by the sub-managers of state s in level $i-1$ then, they will be rewarded. The manager of state s in level i will be rewarded if the action proposed became useful for achieving the command received from its manager of level $i+1$, and so on.

The Kaelbling's approach (`HDG Learning`) [83] also learns to solve complex tasks by decomposing the state space in a similar way. In her work, some landmark states of the state space are selected. These landmarks divide the state space into sets in the following way: for each landmark we define a set composed by the states that are closer to that landmark than to any other landmark. The landmarks can be seen as the higher level of the hierarchy, forming an abstract state space. Each node of the higher level (landmark) has its own subspace, formed by the set of states that are closer to it than to any other landmark. In order to learn, the optimum policy to achieve the goal state is planned in the space defined by the landmark states. The goal state in this level is defined as the landmark that is closest to the real goal state, that is, the landmark that "owns" the real goal state. Once the optimal path is obtained, for each landmark, an optimum policy that leads from the states of the landmark towards the states that belongs to the next landmark (where the next landmark is defined by the policy of the landmark's level) is learned. The union of the policies of all landmark, defines the optimal behavior we are searching for.

Unfortunately, to use those methods it is necessary to give an *a priori* decomposition of the state space (although Kaelbling later claimed that an automatic way to do it could exist). This is not usually easy to do. Also, in `Feudal Q-learning`, it is not clear how to deal with obstacles when they appear in the space state (see [52]).

Another approach related to the above one, consists not in building a hierarchy in the state space, but in the temporal scale. It consists in making backups in different temporal scales (possibly creating a hierarchical temporal structure). Two examples of this approach are found in the Singh's works [167], [168]. Unfortunately, this approach has not been still fully developed (as far as we know). Also, the Sutton's theoretical work [185], [149] is related to this approach and can be useful for developing it.

Behavior hierarchies

This approach consists in the incremental learning of more complex behavior until the original complex task can be learned.

To learn a complex behavior, generally, a recursive procedure composed by three different processes is followed: to decompose the complex task in sub-tasks, to learn a behavior for each of those sub-tasks (recursive step) and, finally, to coordinate the learned behaviors to obtain the complex behavior. This mechanism is considered to be incremental because the abilities of the agent gradually grow in this way.

This method is very similar to the previous approach that is based on building a hierarchy of behaviors from a division of the state space. Some people could even consider them to be in essence the same. In fact, each policy learned that is useful for solving a subset of the state space, can be considered as a behavior needed to learn to solve the complex task. From this point of view, they look like to be the same approach. Nevertheless, we have classified apart these two approaches because we think that there exists a crucial difference between them. The difference consists in that the decomposition of the complex task is made in one case in terms of the state space (that sometimes could be arbitrary), and in the other case is made in terms of the actions used to solve the task (*that usually is more closely related to the structure of the task*).

There are different systems based on this hierarchical idea. Some of the works we are aware in reinforcement learning are [92], [169], [192], [108], [24], [155], [154], [10], [9], [60], [48], [74], [77], [76], [75] and [22], although a lot more work has been done in *supervised* hierarchical learning, for instance in the neural networks field.

All these systems can be classified according to the following parameters:

1. **Kind of coordination:** It indicates the kind of coordination that the complex behavior makes over the sub-behaviors[1]. Usually it consists in a *supervisory* coordination in which, at each time step, the sub-behavior that has to be activated (switch case) is determined. In other cases, the supervisory coordination determines at each time step the weighted influence of all sub-behaviors for the current action to execute (weighted sum). A special kind of coordination is the *delegatory* one in which the control of execution is passed to the sub-behavior and it is returned only when the sub-behavior has finished its work (with success or failure). The delegatory coordination is the most useful kind of coordination in order to speed up the learning process. It simplifies the learning process because it is not necessary for the behavior to take a decision at each time step but only when sub-behaviors return the control to it.

[1] We call sub-behaviors to each of the n behaviors (already learned or not) needed to learn the complex behavior.

2. **Learning by steps:** This feature indicates whether the sub-behaviors have to be learned *a priori* or they can be learned at the same time as the learning of the complex behavior. There also exists the possibility of an incomplete *a priori* learning of sub-behaviors that later are refined while learning the complex task.

3. **Number of reinforcement functions:** This feature indicates whether the learning algorithm needs one local reinforcement function for each sub-behavior to learn or it is enough with only one global reinforcement function. From the point of view of the programmer, it could seem better to have only one global reinforcement function, but this is not always a good idea because having a specific reinforcement function for each behavior speeds the learning of the complex task. In some way, the specific reinforcement functions work as intermediate reinforcements (see section 5.1.1), because they provide information that guides the learning process in addition to the information provided by the global reinforcement function for the complex task. Nevertheless, they are also different from intermediate reinforcements because they are tied to the learning of a specific useful sub-behavior.

4. **State space of subtasks:** Is it necessary, for learning each sub-behavior, to share the same state space (i.e. the same set of perceptions defined by the same set of sensors) or it can be different depending on the case? Of course, it would be more useful in order to speed up the learning process, to focus only on the relevant features for learning each sub-behavior. This would reduce the space of policies in which to search the desired behavior.

5. **Action space of subtasks:** Is it necessary, for learning each sub-behavior, to share the same action space (i.e. the same set of actions) or it can be different depending on the case? It would be better for speeding up the learning process, to focus only on the actions that could be useful for the solving of the task. This would reduce the number of possible choices for each situation and, consequently, it would accelerate the learning process.

Having in mind these parameters, we will now review some of the works made for hierarchical learning that use reinforcement techniques.

One of the first approaches to hierarchical learning is the work of Mahadevan and Connell [108]. They present a reinforcement learning methodology applicable for the subsumption architecture [29]. This architecture can be seen as an instance of a hierarchy of behaviors coordinated by *suppressive links*. These suppressive links are activated by sensorial conditions and inhibit the output of the other behaviors. In this way, they select for each perception the adequate behavior to execute. Unfortunately, in this approach, the supervisory coordination of behaviors implemented by suppressive links has to be *hand-made* and, thus, learning only occurs on sub-behaviors. Learning of the sub-behaviors is performed at the same time, having each one of them its own appropriate reinforcement function.

Another of the first works is the `CQ-L` approach of Singh, developed in [169] and [170], and used for Tham and Prager in [192]. The approach is designed for learning behaviors that can be described as composed of an arbitrary sequence of sub-behaviors. Learning of sub-behaviors and of the complex behavior is done at the same time. Each sub-behavior has its own reinforcement function that returns a positive reinforcement when the goal state for the sub-behavior has been achieved. No reinforcement is given for the complex behavior when the goal of a sub-behavior is achieved (although the achieving of the sub-task is reported to the agent by changing an "on purpose" bit in the perceptual system). A switch gate learns to choose, in a supervisory way, the sub-behavior to use for each perceptual state. One of the merits of the architecture, although it is designed to supervise in each step what sub-behavior to use, is that it is able to learn a sequential composition of them to solve the complex task (that is, it is able to learn a *complete* execution of one sub-behavior, then another one, and so on until the goal of the composite task is achieved). Also, it is able to learn behaviors that are useful to solve a composite task although it has not been never trained for this sub-behavior specifically[2]. On the other hand, `CQ-L` is designed only for learning *pure* composite behaviors, that is, behaviors composed only by the execution of the learned sub-behaviors and do not include other possible actions. Another inconvenience of the approach is that the space of perceptions and the space of actions has to be shared by all sub-behaviors.

Lin's work [92] is also one of the pioneers on hierarchical learning. He proposes to design a hierarchy of skills, to learn them and finally to coordinate them in order to solve more complex problems. His approach uses `Q-learning` to learn all the behaviors. In the mechanism proposed, the coordination is delegatory. In this case, control is passed to the chosen sub-behavior until another sub-behavior becomes applicable. When this happens, a new decision process is performed and the next sub-behavior is chosen.

Each sub-behavior has its own reinforcement function and can be defined in different state and action spaces. They have to be defined by the teacher. The teacher also has to define for each sub-behavior its *application space*, that is, the conditions where the sub-behavior is applicable. This definition is critical, because the space of application of behaviors has to overlap, but not too much. An automatic procedure to determine the space of application for each behavior has been proposed in the same work. It is based on the distance of the current state to the goal state of the behavior to determine whether the state has to belong to the application space, but it does not seem to be applicable in all kinds of tasks (for instance, in the behaviors of the *fruitcake* task; see section 5.5). In this

[2] Although, the architecture has not been designed to perform an automatic decomposition of one behavior in sub-behaviors.

approach, behaviors can be learned separately by steps or they can be roughly learned by steps first and later refined all together at the same time. Finally, this approach is designed to find optimal solutions (`Q-learning` is used), but it does not ensure the finding of optimal paths.

Another approach related to hierarchical learning is represented by the works of Dorigo and Colombetti about *shaping* of animats [48] [60] [49]. The *shaping* term is taken from ethological research and, in their work, makes reference to the training of an agent through an *immediate* reinforcement (that they call *supervised reinforcement*). In their approach, they propose a supervisory coordination in a hierarchical architecture of behaviors that are built from classifier systems. The learning of a complex task consists in making a shallow learning of the basic behaviors, later learning how to coordinate them and, finally, learning all together. The coordination of behaviors can be either by switching to one behavior to be executed, or by weighting the proposed actions of all behaviors to produce a mixed output action. It must be noted that all the work done in this framework is limited to immediate reinforcement and it is not clear how to adapt the proposal to delayed reinforcement.

In hierarchical learning, there is also the Humphrys's `W-learning` algorithm [77] [76] [75]. This algorithm learns to coordinate several different learning modules, where each module learns one behavior useful for the complex task. For each module, there must be a different reinforcement function that expresses the kind of task it tries to solve. For learning, each module proposes at each decision step what it considers to be the best action to execute. The algorithm learns a value table W that indicates, for each perceptual state which is the module (sub-behavior) appropriate to believe. Learning in each module is done by using `Q-learning`. The learning of sub-behaviors and the W table is made concurrently. In order to determine the performance of the agent in a complex task, a global fitness function is used. To obtain an optimum performance for the complex task, a searching mechanism is used so as to find the adequate balance (weight) between the different reinforcement functions for the sub-behaviors. In must be note that each behavior learns in its own state and action spaces, but in some variations of `W-learning` these spaces must be shared.

The Asada *et al.* work [10] [9] is different from the other works described and even someone could consider that it is not a hierarchical approach. The focus of this approach is on robot learning by using previously learned behaviors, as in the other cases. In spite of this, this work does not use the supervisory or delegatory view about coordination but builds a new behavior. This new behavior is defined in the union of the state space of the previous behaviors, reaching a consensus between the previously learned behaviors on the Q-values of the states when the behaviors share the state space, but augmenting it with new perceptual information and learning new Q-values when the previously learned behaviors become incompatible in order to determine the Q-value for a given state. The new behavior built in this way does not use the previously learned behaviors, but is built from them. Obviously, to learn previously the sub-behaviors is necessary for learning the new behavior, and that is why we consider this as a hierarchical approach (hierarchy is not in the architecture but in the learning process). In this approach, learning of sub-behaviors can be done in different action and state spaces. Also, each sub-behavior has defined its own reinforcement function.

Finally, there are some works that usually are considered as hierarchical, but that do not completely fit into our definition of hierarchical learning. As we have seen in section 4.2.6, hierarchical learning has the virtue of easing the perceptual aliasing problem. Thus, *some works use this approach to solve the perceptual aliasing problem, but are not interested in solving the delayed reinforcement problem.*

For instance, Ring's work [155] [154] is usually considered as a hierarchical approach, although we would not classify it as such. The work he presents, has to be considered, from our point of view, in the *incremental-on-the-same-task* approach proposed in section 5.2. Its application has been done for a set of mazes of increasing difficulty, but not for increasing the abilities of the agent in different kinds of tasks. In fact, the hierarchical building of behaviors is mainly made to solve the POMDP problem that appears in those mazes. That is, when for the same perceptual state it is necessary to take two different actions, depending on the current real state, then two different behaviors are built: the first with one action defined for the perceptual state and the second with the other action defined for the same perceptual state (see in section 5.2 the passage titled "Hierarchical approach").

Another instance of using the hierarchical approach to solve the perceptual aliasing problem is the Wiering and Schmidhuber's `HQ-learning` algorithm [208]. In this work, a predefined number of behaviors is used to learn a task. The approach implements an special form of delegatory coordination: the control is sequentially passed from one behavior to the next one. The first behavior is executed until it achieves its goal state, and then passes the control to the next behavior. The next behavior is executed until it achieves its goal state, and then passes the control to the next behavior, and so on until the last behavior is executed. The particularity of the approach is that *it does not only learn the policy for each behavior to achieve the goal state, but also which has to be the goal state for each behavior to allow the solving of the perceptual aliasing.* Thus, the algorithm learns a division of the task into sub-behaviors. If the task is not solved with the proposed division into sub-behaviors (that is, the task is not learned with the proposed goal states for each behavior), the algorithm changes the division proposed until it finds an appropriate division that allows the solving of the problem.

None of these two approaches tries to speed up the learning process of the agent. They are designed to solve the perceptual aliasing problem and then they have to be compared with approaches to solve that problem, for instance `Active Memorization`[3]. (see in section 5.2 the passage titled "Hierarchical approach").

5.4 Our approach: hierarchical development

In this section, our approach to solve the delayed reinforcement problem that is inspired in hierarchical learning, will be shown. This approach is well suited to the CANDID algorithm and to the `Active Memorization` approach.

In chapter 2 and also in the beginning of this chapter, we stated that one of the main difficulties in learning complex tasks is a long delayed reinforcement. In sections 2.2.1 and 2.3, we describe the cause that we consider responsible of a long delayed reinforcement. We redefine there this problem in the following way: *the possibility of learning to solve a complex tasks is subordinated to the set of actions available to the agent. When the task needs the execution of a large number of actions in order to be solved, then we must conclude that the agent's set of actions is not adequate to solve the task.* Thus, to solve the task the agent has to previously learn an adequate set of behaviors that allow it to solve the task in a few steps. In our approach, the agent describes over time a stepped development process [112]. Each step of the development process is characterized by the set of actions of the agent. With the learning of a new behavior, the agent increases its set of actions and becomes more competent for solving more complex tasks.

The mechanism for learning new behaviors that we propose for this approach is based on the incremental approach. The agent uses the set of available actions to learn the new behavior. Then, this new behavior is added to the set of actions of the agent. This increase the abilities of the agent in order to face more complex tasks. In this process, a hierarchy of behaviors is generated, where complex behaviors use more simpler ones as primitive actions.

This development process, based on hierarchical learning, gives the name to our approach: `hierarchical development`.

In order to achieve the needed skills for facing a complex task using our approach, the following recursive procedure composed by three different processes is performed:

1. *Decomposition of the complex task into sub-tasks:* In this phase, the complex task is divided in simpler sub-tasks. This decomposition is made by the teacher.

[3]In a sense, `Active Memorization` works in a similar way that the `HQ-learning` algorithm; when `Active Memorization` uses a memory bit, it learns to split the task into two different policies (behaviors): one for the memory bit set to 1 and another for the memory bit set to 0. Thus, it also learns a division of the task to solve the perceptual aliasing problem.

2. *Learning a behavior for each of the sub-tasks:* In the learning process of each of those sub-tasks, a sub-behavior is generated. These sub-behaviors are learned using the CANDID algorithm and, if necessary, `Active Memorization` from the current set of actions available to the agent. If one sub-task to learn is too complex to be solved with this set of actions, it is decomposed repeating the whole procedure (recursive step). Each new sub-behavior is added to the set of actions of the agent as a new action.

3. *Learning to solve the complex task:* Once the behaviors for all sub-tasks are learned, the agent can face the learning of the original complex task. The task will be learned by using CANDID and `Active Memorization` with a set of actions that includes the behaviors learned in the previous step.

Using this approach, the long delayed problem is circumvented. In each learning process, the agent only faces tasks that are feasible to learn.

In the resulting approach, coordination of sub-behaviors becomes delegatory. This means that control of the execution is passed at each decision step to the chosen sub-behavior, and it only returns the control to the higher level when it has finished with either failure or success. *This delegatory coordination becomes positive* because it eases the learning process by removing the responsibility, on the higher level, of deciding *at each time step* the behavior to be executed. It has to determine the behavior to be executed *only when control is returned to it.*

Behaviors are learned by using the CANDID algorithm and `Active Memorization`. The learning of sub-behaviors is done following the above described procedure until the agent is able to learn the complex initial task. Learning of a new sub-behavior, in addition to the adequate set of actions, requires the following information:

- When the positive reinforcement signal must be given to the agent. In our approach, this is usually equivalent to know when the goal state for the current task has been achieved.

- When the negative reinforcement signal must be given to the agent. In our approach, this is usually equivalent to know when the maximum number of actions allowed to take in order to solve a problem has been exceeded, and to know whether the agent has fallen in a forbidden or dangerous state.

- The subset of useful actions to solve a task from the whole set of possible actions, as well as the subset of sensors.

In order to learn a task, it is obviously necessary to know when the agent has achieved the goal. This knowledge is usually very easy to describe by the teacher.

Note that no other kind of knowledge, that usually is more hard to describe, is required; no progressive approaching to the goal or immediate reinforcement are needed in this approach.

On the other hand, it is also necessary to know when the agent has not succeed in solving the problem. Failure is easy to characterize too. Following the CANDID approach (see chapter 3), it occurs when the maximum number of actions to be executed has been exceeded and the goal has not been reached. Finding the number of actions necessary to solve the task is not usually difficult to guess. Nevertheless, the parameter is not critical in the CANDID algorithm, that means that an error in the estimation of the parameter is not paid with bad policies or with policies that are far from the optimal one (compare, for instance, the number of steps allowed and the maximum number of steps resulted from policies obtained in experiences of section 4.4). Finally, when that estimation cannot be done, there also exist automatic procedures that can guide the learning process to find that value (see section 3.3). Also, a failure signal can be given to the agent when it reaches a forbidden state. The forbidden states must be defined by the teacher. They are useful to avoid the exploration of dangerous states or the exploration of states that the teacher knows would become no profitable to visit.

The two items described so far, define the reinforcement function of the sub-behavior to be learned. *Thus, each sub-behavior has defined its own reinforcement function. As we have seen in the discussion of the parameters that describe each hierarchical approach, this is useful to speed up the learning process.*

The last item of the list eases the learning process by indicating which is the set of useful actions and sensors to solve the task. This reduces the size of the state space where to search for the policy. This knowledge is also usually easy to express, but not always. Unfortunately, it becomes critical in the sense that when the number of actions and sensors is increased, the learning process slows down.

Nevertheless, from the description of this item, it must be noted that a characteristic of our approach is that *all behaviors work in its own space of actions and perceptions. We have seen that it also accelerates the learning process because it allows to focus only on the features and actions that are relevant for the task.*

With the above considerations in mind, we must note that the proposed approach presents all properties that allow an efficient hierarchical learning of a complex task: delegatory coordination, different reinforcement functions for each sub-behavior, and separate spaces of actions and perceptions for each sub-behavior.

Another feature of this approach is that learning is done stepwise. It means that before learning a complex task, it is necessary to learn the adequate sub-behaviors. Nevertheless, in this approach it is also possible to complete the learning of sub-behaviors at the same time that the behavior for the complex task is learned. This is useful for the following case. With CANDID, general and reliable sub-behaviors for a given task are usually learned. In spite of that, the use of a sub-behavior for solving a given complex task is sometimes constrained to a reduced set of initial conditions. It would be interesting, in this case, to *tune* the learned general sub-behavior to the initial conditions where it is used. The procedure we propose works in the following way: first the learning of the sub-behaviors is done; then the learning of the complex behavior is done; *finally, the concurrent learning of the complex behavior and the sub-behaviors is done.* This last process forms the *tuning procedure* of sub-behaviors. This tuning procedure changes the general learned sub-behavior making it more suited to the initial conditions where the sub-behavior will be activated. In this case, the adaptation of the sub-behavior could imply, in part, the lost of its generality. In order to prevent this effect, *the changes learned for the sub-behavior will be only used in the complex task for which the tuning of the sub-behavior was done. That means that the adaptation of the sub-behavior is considered as a part of the behavior for the complex task, but it is not considered as a part of the sub-behavior.* If one sub-behavior learned is necessary for solving another complex task, it will be used in its original and general form.

The tuning procedure could be useful also in the following case. We have seen that learning in CANDID stops when, after several experiences (set as a parameter), the policy does not change (see this parameter in section 3.3). We noted also there that the algorithm could stop the learning process with not a totally reliable behavior due, for instance, to a failure in the sub-behaviors for some special conditions (see the parameter in section 3.3). In this case, the performance of the behavior for the complex task could be improved by tuning the sub-behaviors. See, for an instance of this case, results presented in the next section for task 4.

We consider this approach as the more suited and profitable one for learning complex tasks with embedded agents. There are several reasons for this belief.

First, *the approach is not intrusive for embedded agents.* There is no need for any assumption about the interaction of the agent and the environment, and no *a priori* knowledge of the environment, etc.

Second, the agent learns to solve the sub-tasks proposed just in the same way (by using the same algorithm) that it learns the complex task. Thus, *the hierarchical approach does not add any new condition to the be fulfilled* than the needed when it was decided to use the algorithm to solve the complex task. The only new requirement necessary to use this approach is knowing how to decompose the task in useful subtasks. This must be done by the teacher, and it seems that humans can do it very easily.

Finally, the learned behaviors are useful not only to solve the complex task, but also to solve other kind of tasks. That means that the tasks are not only subordinated to the complex task to learn, but also are *reusable* for solving other tasks. Thus, the learning of new behaviors can be seen as a growing process in the abilities of the agent, not only as a process in the learning of the complex task.

5.5 Experiments: The fruitcake problem

In order to empirically test the performance of `hierarchical development` approach described in the previous section, it will be used to learn a known complex task. The results obtained are described in the this section.

The environment selected for the test is the "blocks world" (already seen in part in section 3.2) that fulfills the required conditions of complexity for embedded agents: complex tasks can be proposed incrementally, the limited perceptual system returns incomplete or ambiguous information about the environment, the initial set of actions is too simple to solve complex tasks, etc. This environment has been used previously by Chapman [34] and Whitehead and Ballard [206] to show the possibilities and drawbacks of reactive systems in complex domains.

Blocks world sometimes has been considered, in some fields of artificial intelligence, as an environment for *toy problems*. This depends on the kind of approach that we use and the resources that are available. Reactive systems has been useful for solving robotic tasks (where other approaches has been unsuccessful), but it has been argued that they would not be useful for other kind of tasks. For instance, Ginsberg stated in [69] that reactive systems would not be useful for tasks that involve a very great number of different states, because it would be impossible to maintain such reactive behavior, that is, in this case it would be impossible to store an action for each different state. In the same paper, Ginsberg proposed, as an instance of this kind of tasks, the *fruitcake problem*[4] defined in the blocks world. Nevertheless, Chapman [34] solved very quickly this problem by using a "deictic" perceptual system that reduces enormously the number of states for the reactive agent. He demonstrated by solving this problem that reactive systems can accomplish complex goals in this domain without the combinatorial explosion of stimulus-action rules predicted by Ginsberg [69].

Chapman showed that reactive systems can solve this kind of tasks *by working with an incomplete description of the world.* Unfortunately, the reactive system he proposed was totally programmed and it was not clear how to allow this kind of agents an automatic learning of the task. This problem becomes more pronounced for reinforcement learning because most algorithm needs a complete information about the environment in order to learn appropriately. Thus, the limits for reactive agents were put in the abilities for *learning* to solve this kind of tasks.

Starting from the Chapman's work, Whitehead and Ballard made a first effort [206] in developing a mechanism to enable *learning* in reactive systems with deictic perception. The mechanism they developed tries to overcome the inadequacy of the `Q-learning` [205] algorithm to learn when it is fed with ambiguous information (see section 4.2.2 for an evaluation).

[4]How to build in the blocks world, from a set of blocks labeled with characters, a stack where the word "fruitcake" can be read.

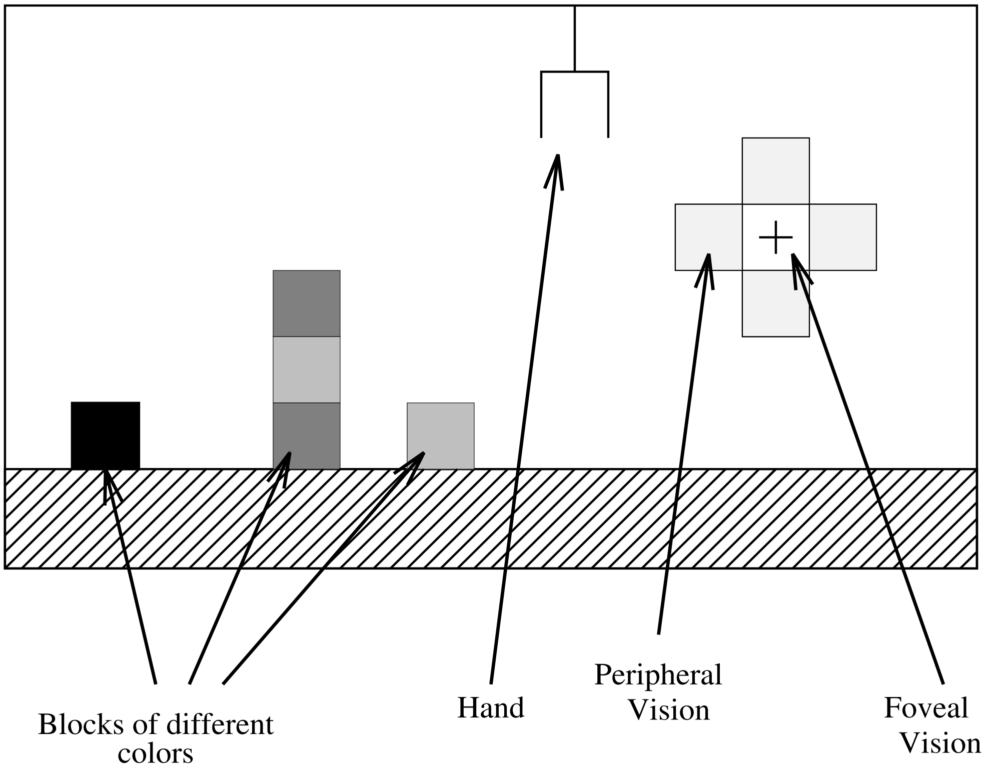


Figure 5.2: A standard setting of the environment.

However, the environments defined by Chapman [34] and Whitehead and Ballard [206] are different from the environment we define. They specify the agent with some powerful complex actions (for instance, visual routines [202]) in the motor system. These actions are programmed as routines and are given to the agent as primitive actions. We do not use such complex actions. In our case, the agent has been designed with a simple set of sensors (that reports an incomplete information of the world) and a very simple set of actions. The set of sensors and actions that we propose, could be physically implemented if necessary. They will be described soon.

Another difference consists in that our environment is not composed of blocks with letters but blocks with colors. This does not imply a qualitative change in the environment or in the task.

In order to perform the experiences, we have developed the `Cooker` simulator of the blocks world. The implementation of our approach for the fruitcake task has been called `Candid cooker`.

A standard setting of the environment (see figure 5.2) is composed of a window where there is a table and blocks of different colors resting on it or forming stacks. In the picture we can also notice the representation of the visual focus of the agent with its peripheral vision, and the hand of the agent. For the sake of simplicity, the environment has been represented as a grid that allows a discrete and coordinated movement of the eye and the hand. This does not imply that the algorithm needs this assumption. We will see in chapter 6 applications of our approach in a continuous environment. If necessary, the current environment could be easily changed also to a continuous representation.

To allow learning of general behaviors, the training of the agent has to take place in different settings. These settings will be randomly generated and presented to the agent as problems to be solved. Each randomly generated setting presents a random number of blocks ranging from 5 to 8. The blocks have a randomly chosen color selected from a set of 4 different colors. To allow the maximum diversity in the structures built with the blocks, the procedure that puts a block in the environment, randomly chooses with a probability of 50% for each possibility, whether the block will be on the table or will be put on another block.

In all settings, the visual focus starts in a random position belonging to an area in the right zone of the window, and blocks are situated on the left zone of the window. The hand is hanging from the ceiling of the world, starting in random free position.

The described environment is the one used for learning all behaviors, except for the last one. In this case, the environment is extended with a *mental model* (see description of final task).

The agent's sets of sensors and actions are the following:

The **Sensory system** is composed of:

- **Visual sensors:**

  - *Foveal vision:* A sensor that represents the visual focus of the agent and gives information about the content of the visualized zone (ground, block (and its color) or void). Thus, this sensor reports only local information about the environment.

  - *Peripheral vision:* Four sensors that convey partial information about the zones surrounding the visual focus. Each sensor indicates only the presence or absence of an object different from void in the zone just to the right, to the left, above or below of the visual focus respectively.

- **Proprioceptive[5] sensors:**
  - *Hand-Fovea angle:* A sensor that reports the relative position between the hand and the visual focus in the world. It can take four values: up-right, up-left, down-right and down-left.
  - *Hand-Fovea distance:* A sensor that expresses the relative distance between the hand and the visual focus in the world. It can take three values: far, near and very-near.
  - *Memory of location (or mark):* A sensor that does not report any information. It is only used to record locations of the environment that are relevant to solve the task. This kind of sensors, with the appropriate visual routines, are used extensively in the Chapman approach [34] to solve the *fruitcake problem*. In our case, it is used with the actions `Move fovea where mark lies` and `Put mark` (see below). Using the first action, the eye can immediately focus the place recorded by the mark (this is why we call it *memory*). The agent has to learn the appropriate use of the mark, that is, it has to learn to remember places of particular interest for the solution of the task. When the visual focus is in one of this places, the agent can record it by executing the second action.

- **Tactile sensors:**
  - *Content:* A sensor that indicates whether the hand is grasping an object or not.
  - *Nearness:* Three sensors that express whether the hand is touching something just to the right, left or bottom of the hand respectively.
  - *Open hand:* A sensor that reports whether the hand is currently open or closed.

- **Memory:**
  - *Last executed action:* An automatic memory that stores the last executed action. It is used to solve the perceptual aliasing problem. Another approach that could be used to solve this problem is *active memorization*. It will be tested in a future set of experiences.

[5] Proprioception is the direct perception of the relative position of parts of the body via, for instance, muscle tension.

Initially, the **Motor system** of the agent is composed of:

- **Actions on sensors:**[6]
  - `Move the visual focus one position`: Four actions that move the visual focus up, down, right or left, respectively.
  - `Move the visual focus to the mark`: An action that moves directly the visual focus to the position where the mark lies.
  - `Put the mark`: An action that keeps in mind the position where the visual focus currently is.

- **Actions on effectors:**
  - `Move the hand`: Four actions that can move the hand one position up, down, left or right, respectively.
  - `Open the hand` and `close the hand`: Two actions that allow to open and close the hand, releasing or grasping and object respectively.

The complex task the agent must solve is the *fruitcake task with colors* that consists in building a stack of blocks forming a predetermined colored structure by using a set of colored blocks randomly distributed. This is the original fruitcake task [34] but substituting letters for colors.

Initially, the agent only has the mentioned set of actions and sensors. In this case, a solution to the *fruitcake problem* will consist of a large chain of actions. Some of the problems that compose this task require to be solved approximately the execution of 200 steps in the arena. This implies a long delayed reinforcement that, as we have seen in section 2.2, is one of the main problems that prevents learning from a *trial-and-error* procedure. In order to solve this problem we will use the `hierarchical development` approach. In this case, the teacher must provide a set of general subtasks to learn which will allow to solve the *fruitcake problem*. For the *fruitcake task with colors*, the teacher proposes to learn the following general subtasks (see figure 5.3):

1. `Searching for a block`: The agent must search in the environment for a block with an specific color. The task is achieved when the visual focus (fovea) of the agent lands on a block of the desired color. The agent has to use its limited visual system and must take advantage of the relevant environmental features (for instance, that any block lies on the table or on other blocks) in order to find the desired block.

[6] Those actions allow what is called the *animate perception* [14] of the agent. It consists in controlling the local sensors in order to find the required information to solve the current task.

2. `Moving the hand to the visual focus`: This task consists in moving the hand towards the visual focus avoiding colliding with any stack of blocks. The visual focus can be on a block or in a void location. In the former case, the block must be cleared.

3. `Removing the top of a stack`: This task consists in removing a block from the top of a stack. In the starting state, the fovea and the hand are placed on the target block. This block must be removed by the hand without colliding with any other block.

4. `Grasping a block`: This task consists in taking a block with the hand. The block can lie at the top of a stack or in the middle of it. In the last case, the agent must unstack previously all the blocks placed upon the target block.

5. `Moving a block to the mark`: This task consists in finding a block with a determined color, in the environment, and putting it where the memory mark is placed.

The learning process generates, for each of these tasks, a behavior that can be considered as a new general action. These new actions increase the agent's repertory of actions and can be used for solving more complex problems. In the following subsections we will show the experimental results obtained applying the CANDID learning mechanism together with our hierarchical learning approach for each of these tasks.

Task 1: Searching for a block

This task consists in searching for a block of a desired color among the whole set of blocks of the world. It has been already presented in section 3.2 as an example for testing the CANDID algorithm.

This is a non-trivial task, due to the limited perceptual information available to the agent. To solve it, the agent must take advantage of the invariant features it finds experimenting in the environment, as the empirical fact that blocks rest on the table or on other blocks.

To make the problem more difficult, only one block of the color we are searching is set in the environment. In order to learn a behavior useful to find blocks of any color, the color of the block we are searching changes randomly after each experience.

In order to solve this task, the whole set of sensors and actions is not necessary. The teacher suggests, for this task, the following set of sensors and actions.

- Useful sensors: *Foveal vision*, *Peripheral vision* and *Last action executed.*

- Useful actions: `Move the visual focus`.

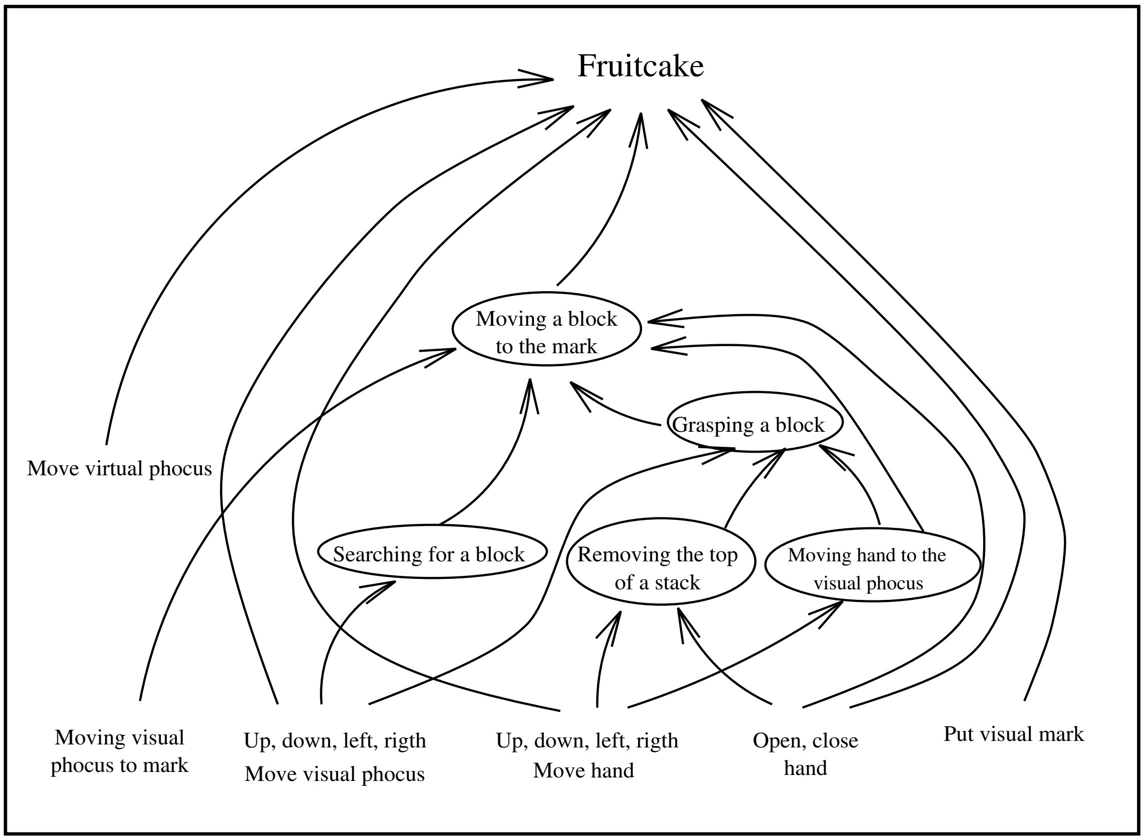


Figure 5.3: Hierarchy of behaviors for the *fruitcake problem*. Intermediate behaviors to solve the *fruitcake problem* are encircled. Arrows represent the used-as-action-to-build-behavior relationship.

The agent will receive a positive reinforcement when it perceives with its foveal vision a block of the desired color, but it will receive a negative reinforcement when it does not reach the goal after 25 actions or when the visual focus disappears from the window of the environment. The maximum number of actions allowed to perform an experience has been set by the teacher. A systematic search for the best value in order to obtain a better behavior, has not been performed.

The learning cycle consists in presenting new randomly generated situations (settings) of the environment to the agent from which to solve the task. The results obtained with CANDID in the `searching for a block` task can be seen in figure 5.4. It represents the number of successes obtained (y axis) for every one hundred settings presented to the agent (x axis). After every 100 experiences presented to the agent for learning, another set of 100 experiences is performed

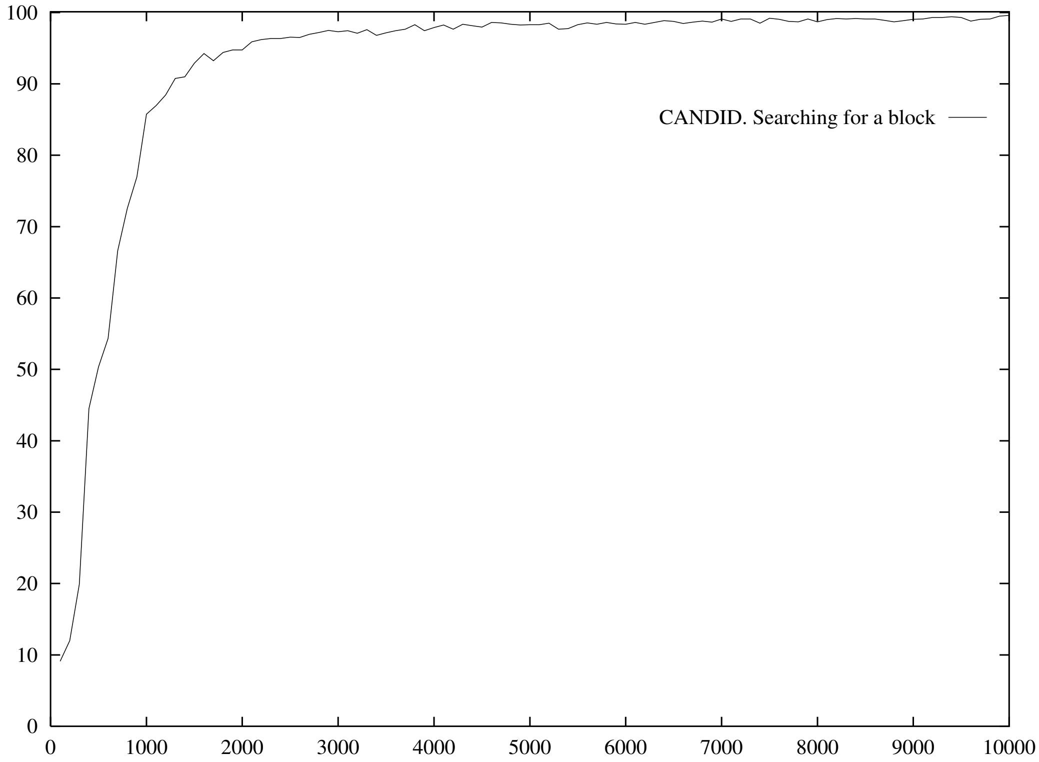


Figure 5.4: Learning curve for the task of `searching for a block`.

for testing the current behavior (ignoring the learning loop of the algorithm). The average of values obtained in the test (that is, after every 100 experiences for learning), for 30 different runs, are shown in the figure.

In the experiences, the value of the N parameter has been set to 100. This is the best value of the four different values of N tested for the task (see section 3.2.3). The maximum number of repetitions has been set to 50.

In order to estimate the difficulty of learning the task, some experiences with random behaviors have been made. In this task, a random behavior achieves 4% of success.

As it can be seen also in the figure, the rate of solved problems using our approach, shows a fast increase at the beginning of the learning: from 1000 experiences on, the agent solves more than 86% of the problems and, from 2000 experiences on, it solves more than 95% of the problems.

We must observe that in all 30 runs of CANDID, the behavior obtained was 100% reliable. CANDID stops the learning process when, after 1000 experiences, the agent does not change its policy. In this case, we can state that a very reliable policy has been learned. In the proposed task, the learning algorithm always stopped, the most expensive run requiring 46400 experiences (although from 12200 experiences on, its reliability was always above 96%), and the less costly run requiring 2300 experiences.

Usually, the learned behavior for this task consists in following the table until a stack of blocks is found; to explore the stack; to return to the table and repeat until the desired block is found. Some variations of this behavior are also learned for unusual configurations of blocks (for instance, when there exists touching stacks of blocks). In the most difficult problems, the agent solves it approximately in 35 steps. Some movies showing the agent solving this task with the learned behaviors are available in the Web at the `http://www-lsi.upc.es/~mmartin/tesis.html` address.

Task 2: Moving the hand to the visual focus

This task consists in learning a general action to coordinate the hand and the eye of the agent. This behavior must be able to move the hand towards the place where the visual focus lies. The hand, as depicted in figure 5.2, hangs from the world ceiling and, thus, it can only reach the visual focus when it is situated on a void location or on a cleared block. While the hand is moving, it must avoid bumping into any stack.

In order to learn this task, the teacher suggests the following as useful sets of sensors and actions:

- Useful sensors: *Nearness*, *Hand-fovea distance*, *Hand-fovea angle* and *Last action executed.*
- Useful actions: `Move the hand`.

The agent will receive a positive reinforcement when the location of the hand coincides with the location where the vision is focused. The agent will receive a negative reinforcement when it does not obtain a positive reinforcement after executing 15 actions, when the hand goes out of the window of the environment, or when the hand collides with any block. The maximum number of actions allowed to perform an experience has been set by the teacher. A systematic search for the best value in order to obtain a better behavior, has not been performed. The estimated number of steps needed to solve the most complex problems of this task is near 25.

The learning cycle consists in presenting new randomly generated situations (settings) of the environment to the agent from which to solve the task.

A first experiment made for this task consisted in presenting different settings to the agent with the learning ability inhibited. In this case, the agent takes actions randomly for solving the problem. The percentage of problems solved using this strategy was around 2%.

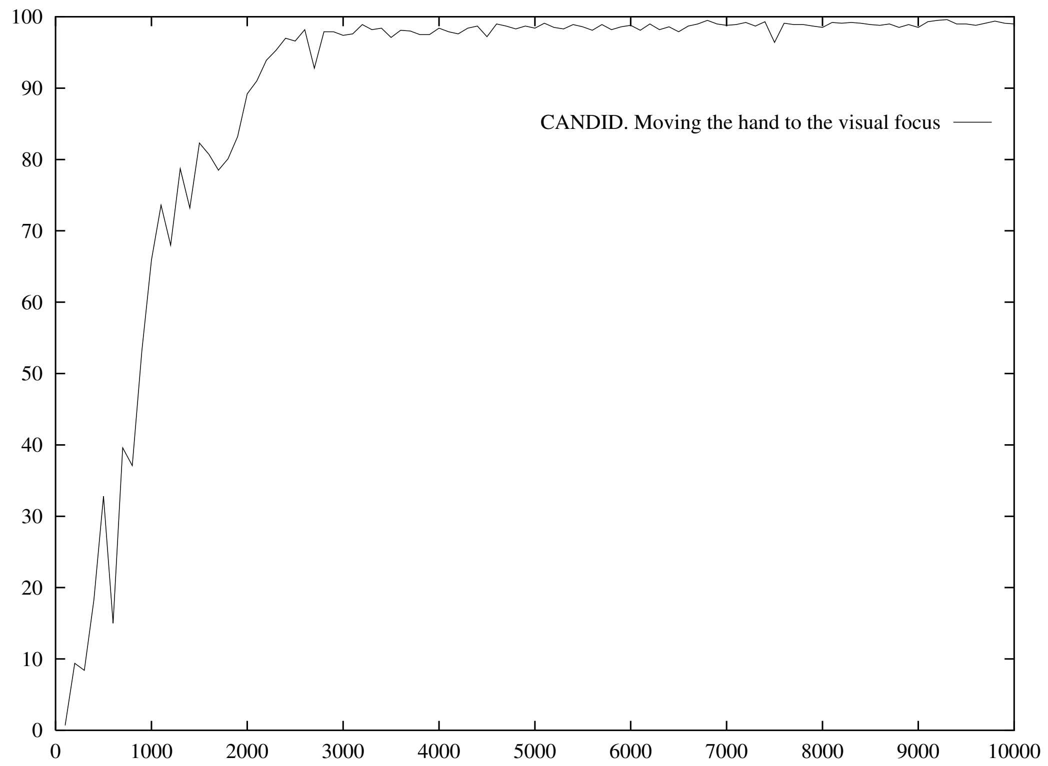


Figure 5.5: Learning curve for the task of `moving the hand to the visual focus`.

The results obtained with Candid for this task can be seen in figure 5.5. This figure has been built following the procedure described in the previous task. The average results for 10 different runs are depicted.

The value of parameter N has been set to 100 by the teacher. The maximum number of repetitions has been set to 50. For neither of the two parameters a search for the best values has been performed.

As it can be seen, results are positive: from 2000 experiences on, the agent solves more than 80% of the problems and, from 3100 experiences on, it solves more than 95% of the problems. In all runs, the obtained behavior was 100% reliable. Candid stops the learning process when after 1000 consecutive experiences the agent did not change its policy. In the proposed task, the algorithm always finishes the learning, the most expensive run requiring 58300 experiences (although from 16800 experiences on, its reliability was always above 95%), and the less costly run requiring 2800 experiences.

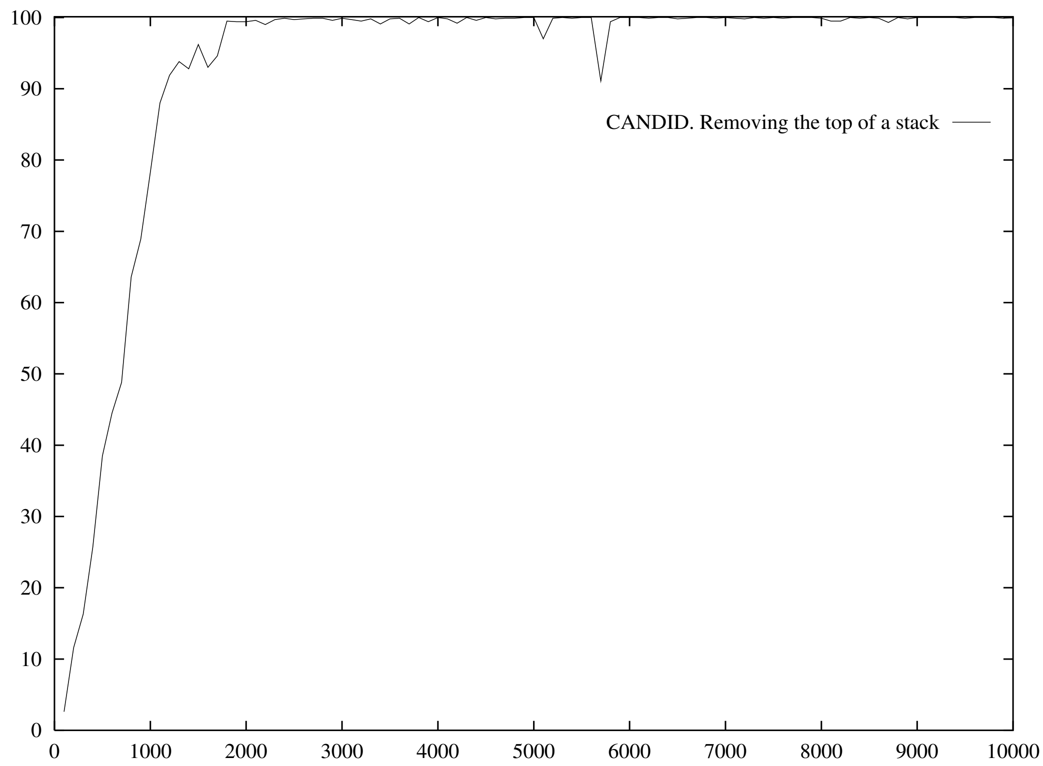


Figure 5.6: Learning curve for the task of `removing the top of a stack`.

Task 3: Removing the top of a stack

This task consists in eliminating a block from the top of a stack. In the initial state, the fovea and the hand are placed on the target block. The block must be removed by the hand, while avoids to collide with any other block.
The teacher suggests the following as useful sensors and actions:

- Useful sensors: *Nearness*, *Open-hand* and *Last action executed*.
- Useful actions: `Move the hand`, `Close the hand` and `Open the hand`.

The agent will receive a positive reinforcement when the block at the top of the stack is removed from its location and left in any other place. The agent will receive a negative reinforcement when it does not obtain a positive reinforcement after executing 8 actions or when the hand collides with any block. Some complex problems of this task are those in which the stack from which the top has to be removed, is located between two higher stacks. In this case, the hand has to move up the target block carefully between the two stacks before leaving it in a free location. The maximum number of actions allowed to perform an experience has been set by the teacher. A systematic search for the best value in order to obtain better behaviors, has not been performed. The estimated number of steps needed to solve the most complex problems of this task is below 15.

The learning cycle consists in presenting new randomly generated situations (settings) of the environment to the agent from which to solve the task.

A first experiment made for this task consisted in presenting different settings to the agent with the learning ability inhibited. In this case, the agent takes actions randomly for solving the problem. The percentage of problems solved using this strategy was around 2%.

The results obtained by CANDID for the task can be seen in figure 5.6. This figure has been built by following the procedure described in the previous tasks. There, the average results for 10 different runs are depicted.

The value of parameter N was set to 100 by the teacher. The maximum number of repetitions was set to 50. For neither of the two parameters a search for the best values has been performed.

As it can be seen in the figure, the task is easily learned by the agent. From 1800 experiences on, the agent solves more than 98% of the problems, except in two cases where two peaks appear. Peaks appear when the agent deals with extremely difficult and infrequent problems. These problems are usually characterized by high stacks of blocks that surrounds the stack from which we want to remove the top. These cases are difficult to solve because it is very easy to collide with other blocks. The repetition of these problems, made to find a reliable behavior, forces a change in the policy, and this produces the peaks. Nevertheless, in all runs performed, the behavior obtained was 100% reliable. The most expensive run needed 11900 experiences (although from 8200 experiences on, its reliability was always above 95%). The less costly run required 800 experiences. In this task, CANDID stops the learning process when, after 2000 consecutive experiences, the agent did not change its policy.

Task 4: Grasping a block.

In this task, the agent must take a given block with the hand. In the initial situation, the visual focus lies on the target block. This is a non-trivial task because the target block can be free of any other block, or be located in the middle of a stack. In the last case, the agent must learn also to unstack the blocks placed above the target block.

In order to learn this task, the following actions and sensors are selected by the teacher.

- Useful sensors: *Foveal vision*, *Peripheral vision*, *Open-hand*, *Content* and *Last action executed.*

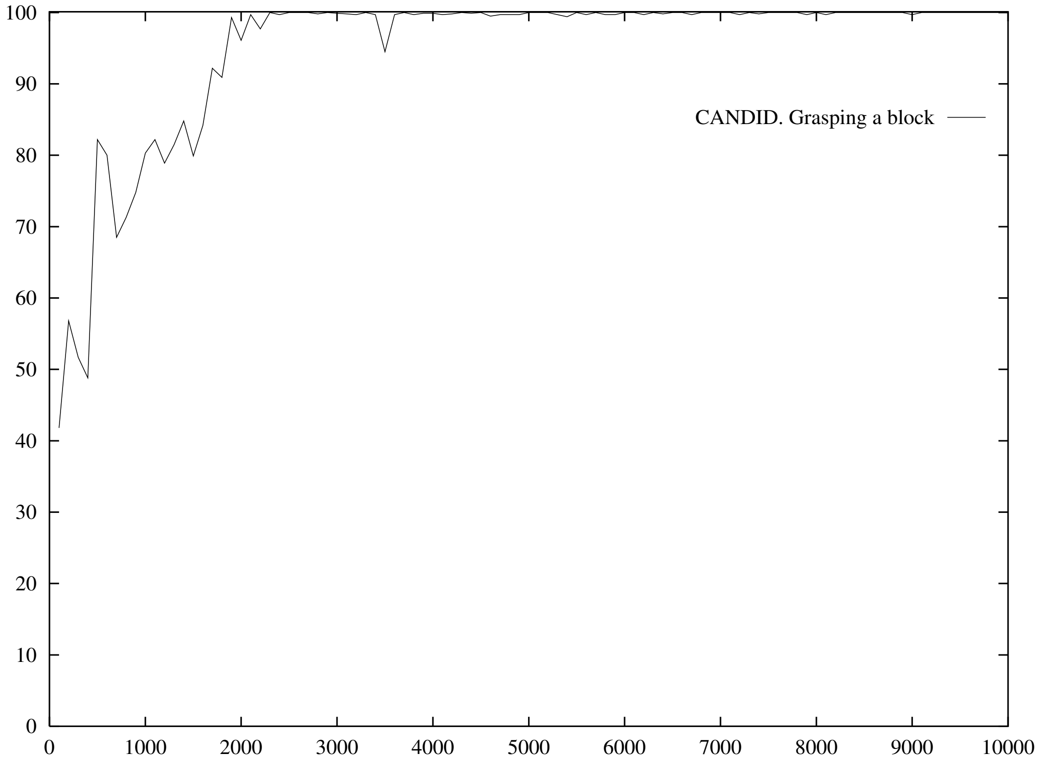


Figure 5.7: Learning curve for the task of `grasping a block`.

- Useful actions: `Move the visual focus`, `Move the hand towards the visual focus` and `Remove the top of a stack`.

The above sets of actions and sensors are suggested only for the `grasping a block` task. The sub-behaviors used for this task (in this case the `Remove the top of a stack` behavior) use their own sets of actions and sensors.

A positive reinforcement will be given to the agent when the desired block is grasped by its hand. The agent will receive a negative reinforcement when it does not obtain a positive reinforcement after executing 18 actions. The maximum number of actions allowed for performing an experience has been set by the teacher. A systematic search for the best value in order to obtain better behaviors, has not been performed. The estimated number of steps needed to solve the most complex problems of this task is below 35. The maximum number of actions allowed and the number of steps needed differ considerably because some actions the agent can take are in fact complex actions built from primitive actions of the agent that involve the execution of several steps in the arena.

The learning cycle consists in presenting randomly generated new situations (settings) of the environment to the agent from which to solve the task. As in the other tasks, a first experiment has been made to determine the performance of a random behavior. For this task, the percentage of problems solved using a random behavior was around 5%.

The results obtained with CANDID for this task can be seen in figure 5.7. This figure has been built following the procedure described in the previous tasks. In the figure, the average results for 10 different runs are shown.

The value of parameter N was set to 100 by the teacher. The maximum number of repetitions was set to 50. For neither of the two parameters a search for the best values has been performed.

As it can be noted in the figure, the task is easily learned by the agent. From 1900 experiences on, the agent solves more than 97% of the problems, except in one case where a peak appears. The peak, as in the previous task, corresponds to the occurrence of a extremely difficult and non frequent problem. Nevertheless, in all runs, the behavior obtained was 100% reliable. The most expensive run needed 9100 experiences (although from 3500 experiences on, its reliability was always above 95%), and the less costly run needed 400 experiences. In this task, CANDID stops the learning process when, after 2000 consecutive experiences, the agent did not change its policy.

This is the first behavior that uses previously learned actions to solve the task. The results for learning the task, without the tuning of the previously learned behaviors, only achieved the 87% of success! The cause of this bad performance is that the sub-behaviors the agent used were not totally reliable. They were highly reliable (they had not failed even one experience from 1000 *random* ones) but, unfortunately, the initial conditions in which the sub-behaviors are used, are not random and for some of this conditions the sub-behaviors fail. Nevertheless, this happens sometimes, but not usually (we will see in the following tasks that this does not occur).

Fortunately, *this problem can be solved with the tuning of sub-behaviors* (described in section 5.4). From near the 500 experiences to the 1500 experiences it is detected that the behavior for the current task does not increase its performance. We can notice it in figure 5.7, where an impasse in the learning curve for this values can be seen. The algorithm detects that the current policy has not been changed in the number of trials that determine when to stop the learning process, but also that the current policy is not totally reliable. In this case, CANDID blames the learned sub-behaviors for the not totally reliable results and enables the tuning of sub-behaviors, that is, it allows the rewarding and punishment of sub-behaviors (with their own reinforcement functions) for chains of actions that produced for solving the sub-problems appeared while solving the higher problem. It could be observed that near the 1500 experiences, the performance begins to increase again and, finally, *a totally reliable behavior is obtained by using the tuning procedure*. The tuning of sub-behaviors allowed, in this case, to improve

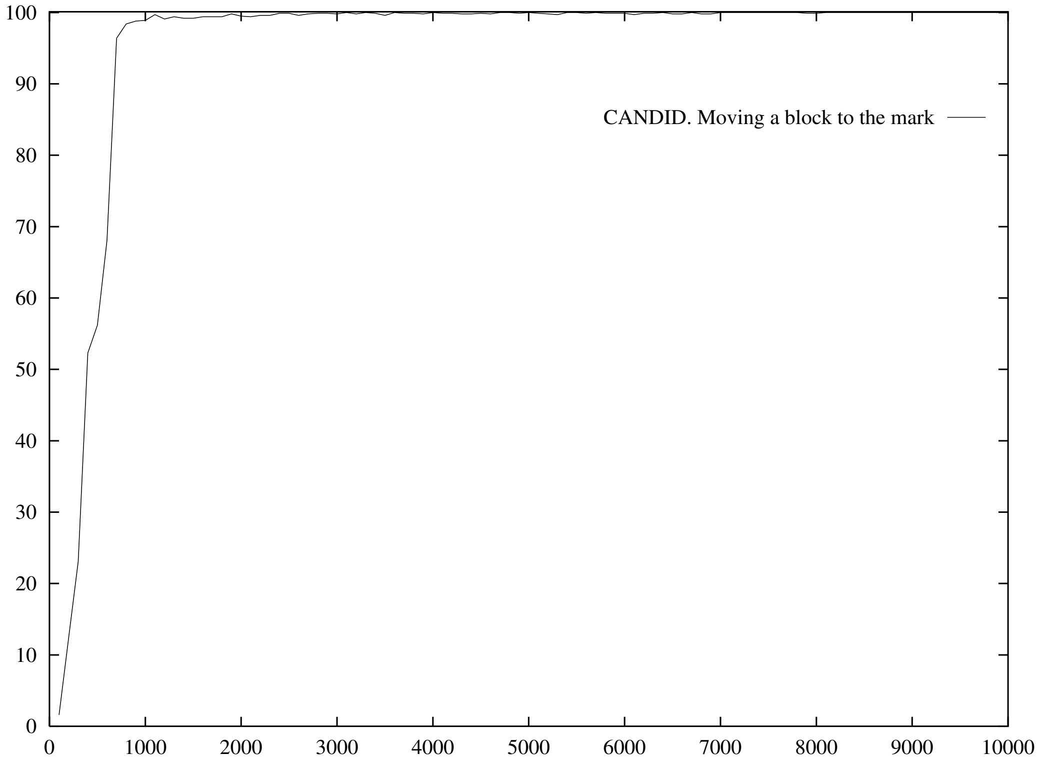


Figure 5.8: Learning curve for the task `moving a block where the visual mark lays.`

the performance up to 100% of success.

Task 5: Moving a block to the mark.

This task consists in moving a block of a desired color to the place where the mark lies. To solve this task, the agent must find the block, grasp it and move it to the target place. The mark is located on an empty location of the ground randomly chosen.
In this case, the teacher suggests the following sets of sensors and actions as the useful ones:

- Useful sensors: *Foveal vision*, *Open-hand*, *Content* and *Last action executed.*
- Useful actions: `Move the visual focus to the mark`, `Search for a block`, `Move the hand towards the visual focus`, `Grasp a block` and `Close the hand`.

These sets of actions and sensors are suggested only for the `Move a block to the mark` task. The sub-behaviors that are used for this task, can use their own sets of actions and sensors.

The agent receives a positive reinforcement when the desired block is located where the mark was initially placed. Note that the mark could not be moved. Thus, the agent has to move the block to the mark in order to obtain a positive reinforcement. The agent receives a negative reinforcement when it does not obtain a positive reinforcement after 7 actions. The maximum number of actions allowed to perform an experience has been set by the teacher. A systematic search for the best value in order to obtain better behaviors, has not been performed. Although the maximum number of actions allowed has been set to 7, the number of steps needed to solve the most complex problems of this task is about 25. This is explained because, as in the previous task, some of the actions the agent can take are complex actions that involve the execution of several steps in the environment.

The learning cycle consists in presenting randomly generated new situations (settings) of the environment to the agent from which to solve the task. In this case, a random behavior can achieve only around 5% of success.

The results obtained by CANDID for this task can be seen in figure 5.8. This figure has been built as in the previous tasks. There, the average results for 10 different runs are shown.

The value of parameter N has been set to 100 by the teacher. The maximum number of repetitions has been set to 50. For neither of the two parameters a search for the best values has been performed.

As it can be seen in the figure, the agent learns easily this task. From 1100 experiences on, the agent solves more than 97% of the problems. In all runs the obtained behavior was 100% reliable. The most expensive run required 8100 experiences (although, from 1800 experiences on, its reliability was always above 95%), and the cheapest run required only 400 experiences. In this task, CANDID stops the learning process when, after 2000 consecutive experiences, the agent did not change its policy.

As in the previous case, the learned behavior uses previously learned actions to solve this task. The results for learning this task without tuning the previously learned behaviors show a 97% of success. In this case, the tuning of sub-behaviors allowed to improve the performance up to a 100% of success.

Final task: color-fruitcake

This is the initial task proposed to the agent. The task consists in building a given stack of blocks. *In order to indicate to the agent how is the colored stack that has to be built, the environment has been extended including a model of this stack.* Figure 5.9 shows the environment extended with this *mental model.* In

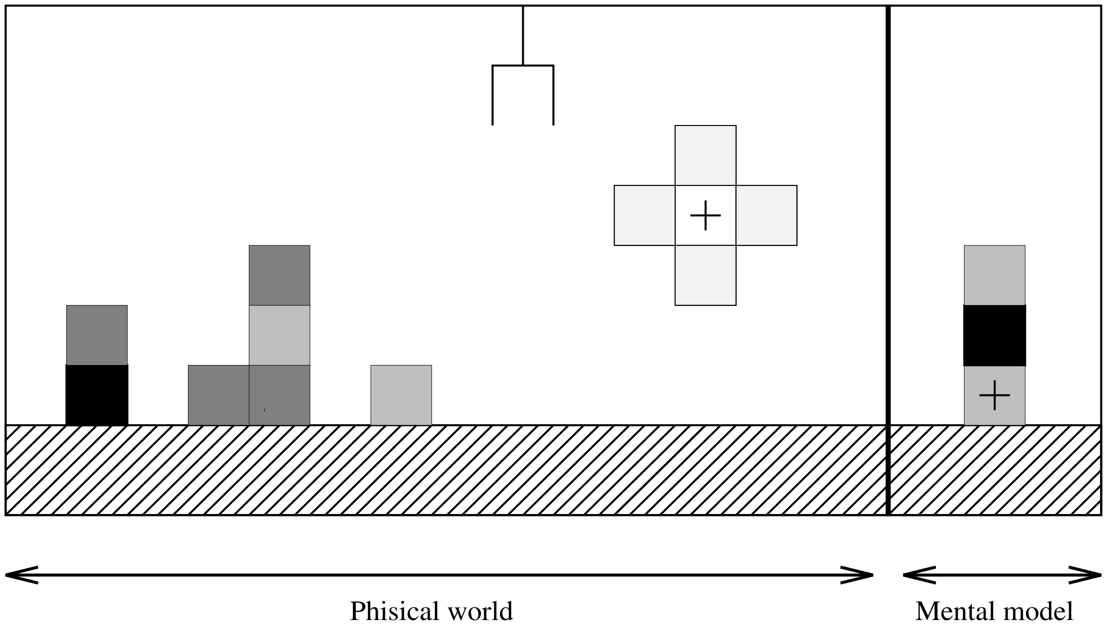


Figure 5.9: Modified environment for solving the fruitcake task

it, the stack that has to be build in the physical world is represented. This kind extension was also used in [34] to solve the fruitcake problem.

The main reason to include this extension to the environment is the following. We are dealing with embedded agent that use deictic representations (see section 2.2.2) of the environment in order to solve problems. Thus, the stack that has to be build, also has to be represented in the same way. In order to allow this kind of representation of the model to be built, the agent is provided with deictic *mental sensors*, that is, sensors that actuate only in the mental model. They will allow the agent to manipulate and assimilate the representation of the target stack. In this way, the mental model includes a *mental fovea* sensor that can be moved up and down with the `Move mental fovea up` and `Move mental fovea down` actions, and that reports the content of the focused zone in the mental model. In the initial state, the mental fovea is always set at the bottom block of the model stack.

For this task, the teacher selects the following actions and sensors:

- Useful sensors: *Foveal vision* and *Last action executed.*
- Useful actions: `Move a block to the mark`, `Open the hand`, `Close the hand`, `Put the mark`, `Move the hand`, `Move the mental fovea` and `Move the visual focus`.

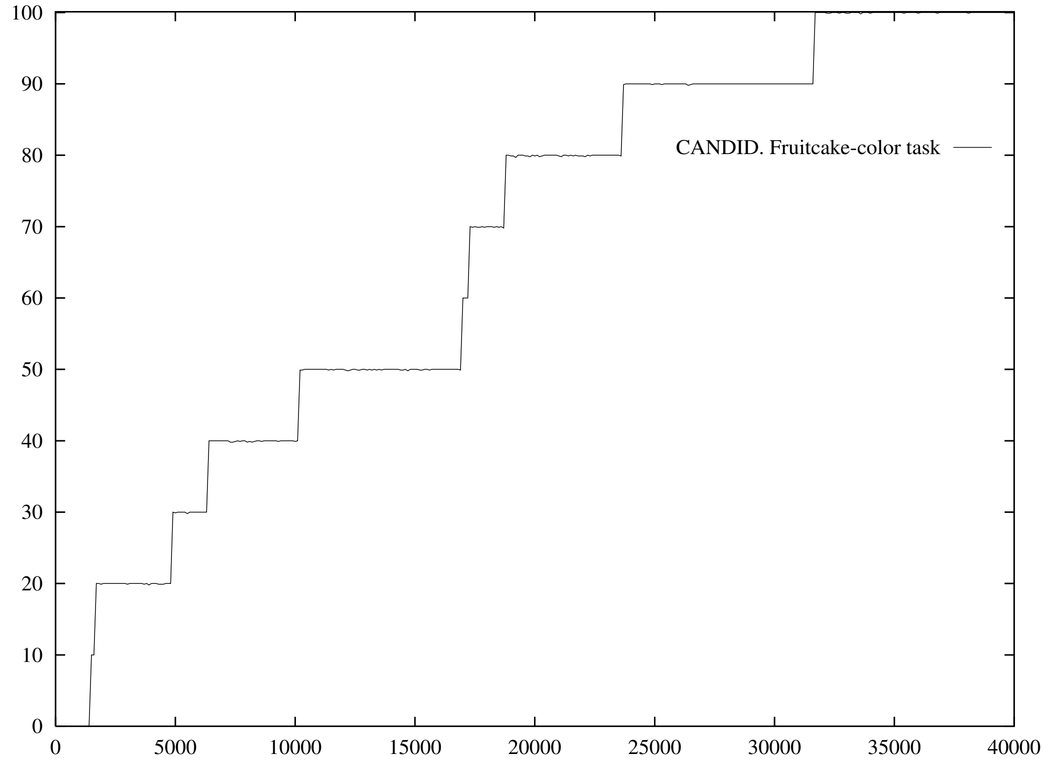


Figure 5.10: Learning curve for the `Fruitcake-color task`.

Unlike the original `searching for a block` behavior, that becomes successful when it founds a block of a randomly chosen color, for the color-fruitcake problem, the `searching for a block` behavior achieves the goal state when it founds a block with the same color that the one of the block where the *mental fovea* lies. With this modification, the behavior searches for a specific block: the block that is indicated by the *mental fovea*. Thus, the `searching for a block` task becomes useful for finding the current block needed to build the stack.

The agent receives a positive reinforcement when the desired stack of blocks is built. It receives a negative reinforcement when it does not obtain a positive reinforcement after 35 actions. The maximum number of actions allowed to execute in a trial has been set without a systematic search for the best value. The number of steps required to solve the most complex problems of this task is a lot higher: about 210. This number of steps required to solve the task for building stacks from only 3 blocks, shows us that this is a really complex task to be learned with reinforcement learning mechanisms, because it shows a long delayed reinforcement. *In the experiments performed, random behaviors cannot* *achieve even one success.*

The results obtained by CANDID for this task can be seen in figure 5.10. The value of parameter N has been set to 100 by the teacher. The maximum number of repetitions has been set to 50. For neither of the two parameters a search for the best values has been performed.

In the figure we can observe a stepped curve. It is the average learning curve obtained for 10 different runs. Each of the runs returned a learning curve of the kind *all-or-nothing*, that is, up to a given number of experiences the agent did not solve any experience and from that given number of experiences on, all experiences are solved. The average of 10 *all-or-nothing* learning curves produces the stepped shape shown in the figure.

In all runs the obtained behavior was 100% reliable. The most expensive run needed 31700 experiences and the cheapest run required only 1500 experiences. In this task, CANDID stopped the learning process when, after 1000 consecutive experiences, the agent did not change its policy.

In this case, the use of heuristics could improve largely the learning performance of CANDID, *because there are fulfilled the conditions for this.* In particular, by using all heuristics together, the algorithm will not repeat the exploration of any policy, at least until a first positive reinforcement is obtained (see section 3.4.2). And when the first positive reinforcement is obtained, then it would be learned to solve any problem because, as we have seen, the learned behaviors for this task are of the kind *all-or-nothing*. In the next version of this work, a set experiences that has been made in order to prove this statement, will be shown.

In the figure, very small oscillations of the of the learning performance in the *plateaus* of the curve, can be observed. These little oscillations are produced by the no complete reliability of the sub-behaviors used to solve the task. The number of failures produced by the no complete reliability of sub-behaviors used, is in this task very small compared with the produced in the previous tasks. The results for learning this task without tuning the previously learned behaviors, shown a 99.7% of success. Also in this case, the tuning of sub-behaviors allows to improve the performance up to 100% of success.

Some movies showing the agent solving this task with the learned behaviors are available in the Web at the `http://www-lsi.upc.es/~mmartin/tesis.html` address.

5.6 Conclusions and future work

In this chapter, we have analyzed the existing approaches to scale up reinforcement learning to learn tasks that show a long delayed reinforcement. In the analysis we have criticized each approach in terms of its possible application to embedded agents with a limited perceptual system for learning general behaviors. Later, we have proposed and tested two different mechanisms to solve this problem. This two approaches are based on the incremental learning approach that proposes a stepwise learning process from the simplest problems/tasks to the more complex problems/tasks.

The first approach is used for the incremental *learning of more complex problems of the same kind of task*. This implies the incremental learning of the same behavior. First the easy problems of the task are presented to the agent. Once they are learned, more complex problems are proposed and so on until the agent has learned to solve the most complex problems of the task with a single behavior.

Experiences performed show that this mechanism can be successfully applied to the CANDID algorithm. The behaviors obtained by using this incremental approach are more efficient and/or more quickly obtained than the behaviors found without using the hierarchical approach. Nevertheless, a more extensive study of the mechanism has to be done.

The second approach to incremental learning that we propose, is based in the hierarchical *learning of new complex tasks* by learning intermediate behaviors. This kind of tasks presents a very long delayed reinforcement when the agent tries to solve them using only its initial set of actions, and (unlike the previous incremental approach) could not be learned by using only the CANDID algorithm. Using the learned intermediate behaviors as primitive actions, the original complex task can be solved in few steps, circumventing then the problem of the long delayed reinforcement. *Our new proposal, called* `hierarchical development`*, allows this kind of hierarchical learning.*

In order to design this proposal, we have done a previous analysis of the features that characterize hierarchical learning approaches, searching for those ones that can make the approach more efficient. The approach that we have developed, presents those features.

The new approach has been tested in a known task: the *fruitcake problem*. This task was tackled in previous researches [34] [206], but the task was only partially learned by them. We have shown by using the `hierarchical development` approach that this task can be learned by embedded agents.

We must note that `hierarchical development` *works well mainly due by the use of the* CANDID *algorithm.* In this approach, the learning of general and reliable behaviors under a limited perception of the environment, is necessary. This condition is ensured by the CANDID algorithm.

Another item that must be remarked in our approach is the tuning mechanism. When the sub-behaviors used in a complex behavior are not totally reliable, the performance of the complex behavior is seriously affected (see the justification in section 2.4.3). To solve this problem, learning (tuning) of sub-behaviors is allowed when the behavior for the complex task does not improve its performance. *It is important to note that the tuning procedure is automatically enabled.* There is no need to indicate when the learning of sub-behaviors has to be begin. As we have seen in the "maximum number of experiences without a change in the policy" parameter in section 3.3, CANDID can detect when the performance of a behavior does not improve; when this happens, the tuning procedure is activated.

Finally, we must remark some features that make our approach efficient: delegatory control, decomposition on sub-behaviors with own reinforcement function, and separate spaces of actions and perception for each sub-behavior.

The bad news are that the teacher must give some information in advance: the decomposition of the task in sub-tasks, and the spaces of actions and perceptions for each sub-task. Although it seems that human teachers can do it easily, in some cases (for instance, in totally autonomous agents or in *animats*) it would be preferable to obtain this information automatically. There exist mechanisms that probably could do that.

One possible way to automatically find the decomposition of a task in sub-tasks and the set of actions and perceptions relevant for the each sub-task, is by using genetic algorithms [115]. In this case, the genetic algorithm would have to find an ordered set of reinforcement functions. Each reinforcement function describes a task to be learned, and the order in the set describes the sequence in which the tasks must be learned. With each of these reinforcement functions, the genetic algorithm also would have to propose a set of useful actions and perceptions to learn the sub-task.

All this information would define an individual of the population maintained by the genetic algorithm [70]. This is the necessary information to use the `hierarchical development` approach. Then, this approach would be applied for all individuals of the population, and the results would be evaluated with an adequate fitness function. From this evaluation, the genetic algorithm would generate the new population and the procedure would be repeated. Finally, the genetic algorithm would return, after several generations, one individual that maximizes the fitness function (that is, a list of sub-tasks and spaces of actions and perceptions for each sub-task, that maximizes the fitness function).

Nevertheless, this is not a trivial application of genetic algorithms because the global fitness function used to evaluate the proposals has to be designed in order not to produce a zero return in any of the failed executions of `hierarchical development`. The fitness function has to differentiate among failed executions, those ones that are closer to solve the complex task.

Other approaches that successfully develop a decomposition of a task in general sub-behaviors, are not useful for our approach. For instance, Thrun and Schwartz's work [200] presents an algorithm that automatically learns general behaviors useful for a task. Unfortunately, in this approach it is necessary to know how to solve the task in order to find these general behaviors. This is not feasible in our approach because the complex task we try to solve is impossible to learn without those sub-behaviors. Thus, these approaches are not useful for speeding up the learning of a task with a long delayed reinforcement, but for the *transfer* of learned behaviors between tasks.

In any case, the price for obtaining automatically the information provided by the teacher, consists in an impoverishment in the efficiency of the approach. That is, in order to find this information it would be necessary to perform a lot more experiences (such number of experiences that, almost for all practical applications, the approach would become inapplicable).

Chapter 6

Application to a realistic robot simulator

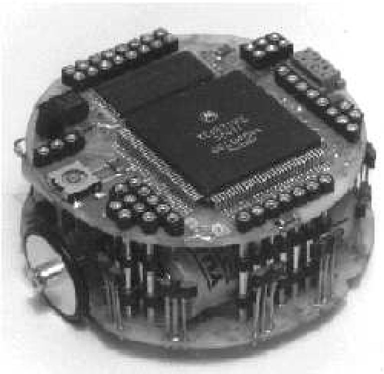
6.1 Introduction

In the previous chapters, we have proposed new methods for learning reliable general behaviors under perceptual aliasing circumstances. Nevertheless, these methods have been tested in simulated environments that, while used in the reinforcement learning literature, are too simplistic. In this chapter we will see how those proposed mechanisms can be applied to *realistic* robotic tasks. These tasks present some features that sometimes are not taken into account in other domains. Two of these features are *noise* and a *continuous environment*.

In order to show the performance of CANDID dealing with these new features, first we describe the robot and the realistic simulator of it that are used in the experiments. Then, we define and justify the task to be learned by the robot using CANDID and `Active Memorization`. Later, we describe the results obtained in the experiments using those mechanisms.

These results obtained are acceptable, but in some cases at the expense of performing a large number of experiences to learn the task or at the expense of obtaining not totally reliable behaviors. In order to improve the obtained results, we make a twofold proposal: on one hand an *incremental approach* designed to reduce the number of experiences spent on learning the task, on the other hand a procedure, that we call *extra-learning*, designed to improve the reliability of the learned behaviors.

Finally, we discuss a surprising result derived from the experiments. We have noticed that one feature that was not present in the experiments described in the previous chapters, *noise*, produces a beneficial effect in the learning of sequential behaviors. We have detected a side effect of noise that facilitates the learning process by performing a subtle kind of generalization that must be more deeply studied.



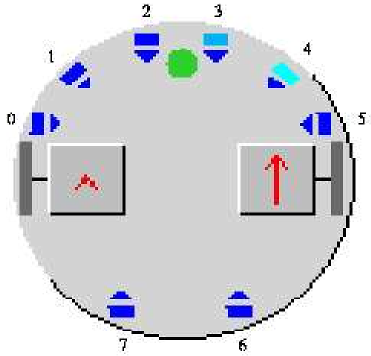


Figure 6.1: Pictures of the real Khepera robot and its representation in the simulator (taken from [135]).

6.2 Description of the Khepera robot and simulator

Khepera [135] is a small robot that has been developed as a tool to facilitate research in control algorithms for robots. It has been used in a lot of works, mainly in evolutionary robotics (see for instance [67] and [161]).

The robot has 8 infrared sensors that allow to detect by reflection the proximity of an object near it. These sensors are located around the robot as depicted in figure 6.1.

The mobility of the robot is granted by two independent motors each one controlling a different wheel of the robot. Each wheel can go back and forward at different speed independently of the other wheel. Thus, the robot can turn by giving different speed values to the motors[1].

In order to make easier the experimentation with the robot, a very realistic simulator has been developed by Olivier Michel [132][2]. One interesting feature of this simulator is that it is able to control the real Khepera robot. Thus, the simulator results can be transfered to the real robot in order to test the learned behaviors.

The simulator shows a window organized in two panels, one of them representing the physical world where the robot is, and the other one showing information for monitoring the state of the robot.

[1] More information about the robot can be found at the following Web address: http://lamiwww.epfl.ch/Khepera/.

[2] The Khepera simulator is available at the following Web address: http://diwww.epfl.ch/lami/team/michel/khep-sim/.

The panel of the world represents an arena of 1 *meter* x 1 *meter* (the diameter of the `Khepera` robot is about 5 cm) in which the robot operates. In this arena, bricks, lights and infrared sources can be placed by the user in order to build the environment of the robot. The bricks are generally used to build complex obstacles or walls in the arena.

The panel used to monitor the state of the robot shows the readings of sensors and current values of actuators of the simulated robot. A representation of the simulated robot is depicted in the right side of figure 6.1. Each of the 8 small rectangles around the robot represents the corresponding infrared sensors of the real `Khepera` robot. Each sensor reports an integer value between 0 and 1023 that indicates the proximity of an object to it. In the same figure the two motors that control the wheels of the robot can also be observed. Each motor can take a value between 10 and -10, that represents the commanded speed to the wheel under its control.

Realistic features not present in previous experiences

Reinforcement learning has been applied successfully in robotic applications, for instance in [54], [108], [92], [9] and [159]. Robotic applications are interesting because they present some features that sometimes are not taken into account in other kinds of applications. Two of these features are *noise* and the *continuity of the environment*.

Noise is a feature present in robotic applications at two levels: at the perception level and at the action level. Perceptions and actions can be noisy for different reasons, for instance, because of poor and non-reliable sensors or actuators, or due to a partial malfunctioning of sensors or actuators.

Noise is present in the `Khepera` simulator both in perceptions and in actions. On one hand, the output value for each infrared sensor is computed as a function of the presence (or the absence) of obstacles, but a random noise corresponding to $\pm 10\%$ of its amplitude is added to the value computed. On the other hand, the robot moves accordingly to the speed set by the control algorithm to each motor, but it is modified by a random noise of $\pm 10\%$ added to the amplitude of the motor speed and a random noise of $\pm 5\%$ added to the direction resulting from the difference of the speeds of the motors.

A robotic environment is continuous in two dimensions: time and space. The concept of continuous environments is opposed to the concept of discontinuous environments (that are usually represented by a grid). In this sense, the `Khepera simulator` is adequate because it is able to represent quasi-continuous environments.

On one hand, the high level of detail of the environment (1 *mm*) with respect to the diameter of the robot (55 *mm*) implies a quasi-continuous *spatial* environment. On the other hand, the movement of the robot in a tick of time, for instance in

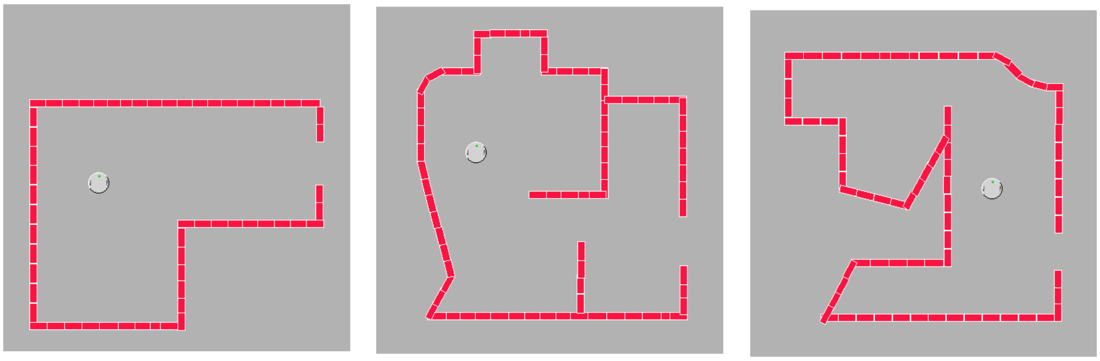


Figure 6.2: Three different rooms used in the learning process of the robot.

the forward action (10 *mm* maximum, 5 *mm* in the experiments), is very small compared to the size of the robot. This implies in practice a quasi-continuous *temporal* environment. This can be noticed because the movement of the robot in the world seems continuous (and not by impulses). In a temporal continuous environment, there are no ticks of time but a continuous flow.

All these features make the `Khepera simulator` so realistic that it can also be used to control the real `Khepera` robot. The simulator can be connected by a serial port to the real `Khepera` robot so that it can be controlled with the behaviors learned in the simulator.

6.3 Description of the task

The task that we propose to be learned with the `Khepera simulator` is to *go out of any room*. This behavior is widely used in navigation tasks.

The robot is initially located in a random position of the room. The goal of this task consists in passing through the open door of the room. Figure 6.2 shows three different rooms designed to train the robot. In all these rooms, a gap in a wall that represents the exit of the room can be observed[3]. All rooms are empty: there are no objects or inner walls. The first room is easier to solve than the other two. Notice that the last two rooms present some corners that are not right angles. Also notice the curved walls and the projecting edges. These features make these rooms specially difficult to solve.

In order to solve this task, the robot must be able to find the wall of the room, to follow it without losing contact with it (and if it loses the wall, to recover it as soon as possible), to deal with the corners of the room, and finally when the robot finds an open door, to go out of the room through it.

[3]Notice that the robot does not have compass information, and thus it is not relevant that in all rooms shown the exit is placed at the left side of the figure.

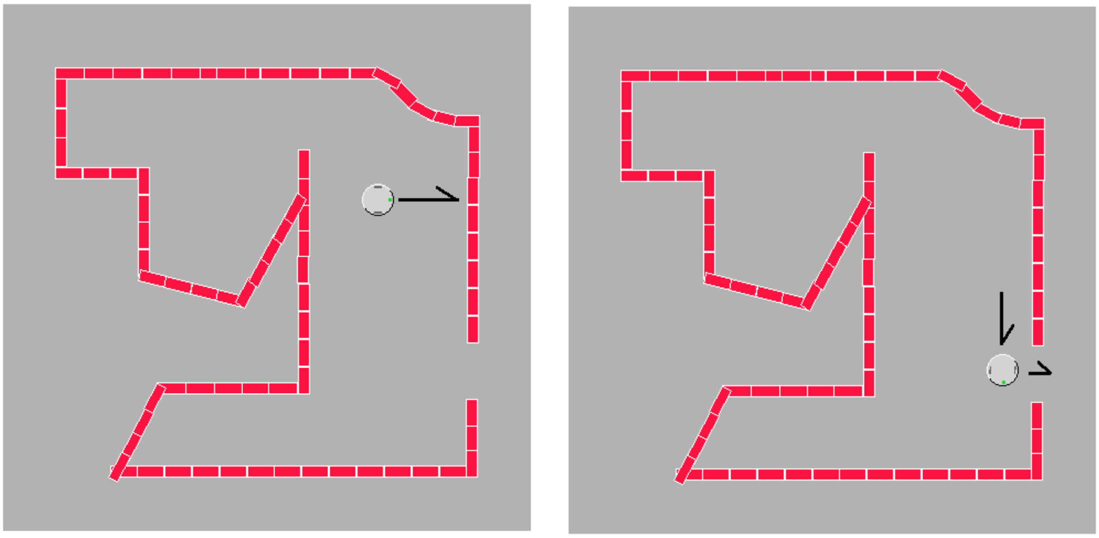


Figure 6.3: Different actions for the same perception. See explanation on the text.

All these abilities have to be learned given the perceptual system of the robot. We must note that, as in experiments shown in previous chapters, the perceptual system of the robot reports local information and that it becomes insufficient to determine unambiguously the real state of the robot in the arena.

In particular, in order to learn this behavior, a mechanism to diminish the perceptual aliasing problem is needed. There are different situations in the process of solving the task that present the same perceptual state for the perceptual system of the robot but that need different actions to execute for solving the task. For instance, in the most common initial situations, the robot usually does not perceive any wall near it, because it is located far from any wall (see the left snapshot in figure 6.3). In this case, the robot must find the wall and the appropriate action is `going forward`. Nevertheless, later, after following the wall (by going forward while the wall is perceived with the right sensors of the robot), the robot will find the door. It is detected when, suddenly, the robot stops to feel anything in its right side (see the right snapshot in figure 6.3). In this case, the robot also does not perceive any wall, but the action to execute this time is `turning to the right`. *Thus, some times it is necessary to go forward and some times to turn right for the same perceptual state.* In order to diminish the perceptual aliasing, we will provide the robot with a memory bit and a memorization action. In this way, we test the `Active Memorization` approach (see chapter 4) also for this robotic task.

To solve this task, we allow the robot to perform a maximum of 2500 steps and 800 actions per trial in the rooms described above. In CANDID, one action is executed until the perceptual state of the robot changes and, thus, the execution of one action can take more than one step. The robot is punished when it has not achieved the goal in that number of steps or actions, when it collides with any obstacle or, finally, when it does not change its position or orientation (significantly) in a predetermined number of steps (for instance 150 steps).

The first case for punishing is needed by the CANDID algorithm in order to obtain acceptable policies (see sections 3.1 and 3.3). The maximum number of steps and actions allowed is a lot larger than the ones allowed in previous experiments, because of the small variation of the location of the robot that each motor action produces in this task. Such large number of steps is required in order to solve the task in the worst case. It must be noted that this number of steps implies, for some cases, a long delayed reinforcement. This will be discussed in the next section and in section 6.6.

The second case for punishing becomes necessary in order to obtain policies that solve the problem without bumping into the walls. Nevertheless, we must note that noise makes difficult the achieving of a policy that never collides with walls, because in case we find one that has defined the right action to execute for each perceptual state, noisy readings of sensors can mislead the robot and produce some bumps. This issue is more detailed in the conclusions of section 6.5.1. In some tests, whenever we explicitly mention it, we will allow a low number of collisions. If the robot is able to solve the problem with a low number of collisions, then they were probably produced by noise (not by the policy). In this way, the performance of the learned behaviors that does not take into account collisions caused by noise, is obtained.

Finally, the last case for punishing is proposed in order to improve the efficiency of the learning process, not in the number of trials but in the number of steps. When the robot does not change significantly its position or orientation in a predetermined number of steps, the trial is stopped with a failure signal to indicate that the robot is blocked in a location from which it does not know how to escape. In this case, we assume that the robot will not change its location even in the case that we let him spend the maximum number of steps allowed. Thus, we stop the trial to save this number of steps. However, as the aim of this case for punishing is to prevent the useless execution of some steps in the *learning* process, it will not be considered in the *test* of behaviors.

6.4 Adaptation of CANDID to robotic tasks

We will use the CANDID algorithm discused in chapter 3 to learn the task described in the previous section. As we have seen in section 6.2, the environment in which CANDID must learn the task presents some new features. In this section we will see how this new features affect the algorithm and how it could be adapted (if necessary) to them.

In fact, CANDID only presents one minor variation to face this task, compared with the algorithm described in chapter 3. Nevertheless, this change does not affect the philosophy of the algorithm and does not vary the results obtained in the experiments described in the previous chapters.

As we can see in the description of CANDID in section 3.1, the agent executes the chosen action for the perceived state until the perception changes. In the previous experiments, the repeated execution of one action always changes (sooner or later) the perceptual state. In the experiments with the `Khepera` robot, this is not the case. It could happen that the repeated execution of one action would not change the perceptual state of the robot. For instance, when the agent is far from any obstacle, the `turn right` action will never change the perceptual state. In this case, the agent would be executing the same action forever, wheeling to the right without noticing any change in its perceptual state.

To avoid this problem, we also reconsider the action to execute when, after a predetermined number of repetitions of the same action, the perceptual state has not changed. In this case, the number of actions taken is incremented and a new action is chosen (which, if the perceptual state has not changed, following the policy, will be the same executed action until now). This modification avoids the unlimited repetition of one action because, in the worst case, it limits the number of repetitions of one action to the mathematical product `maximum number of steps allowed per action` by `maximum number of actions per experience`.

Nevertheless, we must note that this variation maintains the criterion of following always the current policy. The reconsideration of the action to execute does not imply changing the action to be executed if the perception has not changed. This variation is included only to avoid infinite loops while solving a problem. We must note that this variation does not change the results obtained in the previous experiments and that it can be assumed in the original algorithm without changing its philosophy.

One new feature that CANDID must face in these experiments (that was not present in the previous ones) is *noise*. Nevertheless, CANDID *has not been changed in order to face noise*. However, we must note that some conditions initially assumed by CANDID are no longer fulfilled. For instance, noise implies that the repetition of one experience with the same policy and initial conditions could produce *different* results. This affects one of the mechanisms that used to obtain reliable behaviors: the *repetition of failed experiences* (see sections 3.1 and 3.3).

Finally, CANDID *has not been changed in order to face a continuous environment* (the other new feature introduced in this task). However, the *continuity of the environment* implies that a long chain of actions must be executed to solve a problem. As we have observed before, the environment is not really continuous but quasi-continuous because of the high level of description of the simulator and the small effects of a single motor action execution (step) on the location and orientation of the robot. The immediate consequence of using such small steps is that a a large number of these steps becomes necessary to solve a typical problem (a lot larger than in the experiments described in the previous chapters), which implies a long delayed reinforcement. In the experiments performed for this task, a large experience takes about 450 actions (for the first robot; see next section) or 600 actions (for the second robot; see next section) and about 2400 robot steps. This implies a *long delayed reinforcement* that could prevent learning of the behavior. Fortunately, heuristics used by CANDID alleviate this problem. When heuristics H1 and H2 (described in section 3.4.2) are used, the number of *updates* to be made is significantly reduced from the length of the chain of actions to the number of different states that appear in that chain of actions. In the examples proposed above, this number of updates is reduced from 450 to 35 and from 600 to 55 (using the first and second robot respectively; see next section).

6.5 Experiments

In this section we will describe the experiences performed using CANDID and `Active Memorization`. The task to be learned using the `Khepera simulator`, *going out of a room*, has been described in section 6.3.

Perceptual system of the robots

Firstly, we will describe the robots used in the experiments. We have seen that the `Khepera` robot has 8 infrared sensors. Each sensor returns a value ranging from 0 to 1023. This leads to a very large state space (8^{1023} states) that cannot be managed with reinforcement learning mechanisms that use look-up tables. In order to solve this problem we consider three possible simplifications.

The first one consists in defining a threshold of perception that must be exceeded in order to consider a sensor as activated. This threshold has been set to 500, that is, an intermediate value between the two possible extremes. We have not searched for the best value for the threshold. When the value returned by a sensor is above the threshold, then the sensor will be considered on, otherwise it will be considered off. With this simplification, each sensor can take only two possible values (`on` and `off`), reducing in this way the state space of the robot to (8^2 states. We must note that if the resulting robot were not able to solve the task with this modification, we could extend the number of possible values to three (`on`, `indeterminate` and `off`) by defining two different thresholds, to four values by defining three different thresholds, and so on, until the robot were able to solve the task.

The second possible simplification in order to reduce somehow the state space of the robot will consist in ignoring the back sensors of the robot (sensors *S7* and *S8*). Initially, they seem not necessary to solve the task. As in the previous simplification, if the resulting robot were not able to solve the task, we would consider to add them to the perceptual state.

Finally, to reduce the state space of the robot, we will consider the possibility of grouping the information of some sensors of the robot into one single virtual sensor. The resulting robot will be characterized in this case by the number of virtual sensors (that will define the perceptual state), the real sensors of the robot that compose each virtual sensor, and the function used to integrate the information of the real sensors into one output of the virtual sensor.

Considering these possible simplification, two different robots will be used to perform the experiments. The first robot (`R1`) is characterized by 3 virtual sensors that group respectively real sensors of `Khepera` *S1*-*S2*, *S3*-*S4*, and *S5*-*S6*. Real sensors *S6* and *S7* will be ignored.

The information reported by the real sensors is converted into the output of the virtual sensors in the following way: first, the value reported by each real sensors is compared with the threshold in order to determine whether the real sensors are `on` or `off`; later, the following function is applied to calculate the output of the virtual sensor number i (VS_i):

$$Output(VS_i) = (RS_{2i-1} \vee RS_{2i}) + (RS_{2i-1} \wedge RS_{2i}) \tag{6.1}$$

where RS_j returns 1 when the real sensor number j is `on`, and 0 otherwise.

In other words, a virtual sensor will return 0 if the two sensor grouped by it are off, it will will return 1 when only one of its real sensors is on, and finally, it will return 2 when the two real sensors are on. As we can note, each virtual sensor becomes three-valued. Thus, the resulting robot will have 3 three-valued virtual sensors.

However, we will use also in the experiments a second robot (`R2`) that does not group real sensors in virtual sensors. The only preprocessing of the information is made in order to determine whether each real sensor is above or below the threshold. As in the previous case, sensors *S6* and *S7* are ignored. Thus, in this case we have a robot with 6 two-valued sensors.

We have considered other possible configurations of the robot, but some of them were so simplistic that the robot was not able to solve the task. For instance, we have considered a robot with only 2 virtual sensors with values that correspond with the coded information of real sensors *S1*, *S2* and *S3* on one hand, and sensors *S4*, *S5* and *S6* on the other. However experiments made with this robot showed that it was not able to solve the task *without systematically bumping into* the walls of the rooms.

Finally, we must remark that the robots designed has neither compass information about its orientation in the room, nor computational models of the current room.

Motor system of the robots

The `Khepera` robot has 2 independent motors. Each of them controls a different wheel of the robot. Each motor can take an integer value ranging form -10 to 10. Thus, the number of possible actions of the robot is 21x21=441. This is, again, a too large number of actions to be managed with reinforcement learning mechanisms that use look-up tables. Like in the case of the perceptual system of the robot, in order to facilitate the learning process, we will reduce the number of motor actions the robot can take. In the experiences, the robot will have available only 3 motor actions: `rotate-left`, `rotate-right` and `move-forward`. These actions are implemented by giving respectively the $\{-5,5\}$, $\{5,-5\}$ and $\{5,5\}$ values to the motors of the robot (the first value for the left motor and the second for the right one). These actions should be enough to allow the robot to solve the task.

Characterization of the task

The resulted characterization of the task for each of the robots described in the previous section is resumed in table 6.1. There, for each robot that we will use in the experiences (represented by the *number of sensors* and the *grouping function* used), are shown the number of values for each virtual sensor, whether it uses memory or not, and from this information, the number of perceptual states and the number of possible policies.

Robot	N.sens.	Group. funct.	N.vals.	Memory	Num.states	Num.policies
`R1`	3	function 6.1	3	Yes	$3^3\,2 = 54$	$4^{54} \approx 3.2\,10^{32}$
`R2`	6	–	2	Yes	$2^6\,2 = 128$	$4^{128} \approx 1.2\,10^{77}$

Table 6.1: Robots used in experiences.

We must note that in this task the robot has a the limited perception of the environment. In Markovian tasks, the number of states represents an adequate estimation of the complexity of the task, but in tasks with perceptual aliasing, the real states become aliased in perceptual states and, thus, the number of states become a not adequate index for measuring the complexity of this kind of tasks. In order to express the complexity of these tasks, we prefer to use the number of possible policies.

As can be noticed from this table, this task presents more perceptual states and a higher number of possible policies than the other tasks faced in the previous chapters.

6.5.1 Experiments and results

The learning of the *going out of a room* task, in this first experiment, is performed using CANDID and `Active Memorization I` with one memory bit.

The task to be learned has been described above (see section 6.3). The learning process consists (as in previous experiments using CANDID) in the presentation of randomly generated problems to the robot until the learning of the behavior has been finished. After each experience the robot updates its behavior depending on the final reinforcement signal obtained.

A random problem is built by randomly choosing a room from the set of rooms available for training (in our case the three rooms described in section 6.3) and by placing the robot inside the selected room in a random location with a random orientation. As we stated in the description of CANDID, the same random problem will be presented again to the robot (in a new trial) when it fails to solve it. When the robot has solved the problem or when a predetermined number of repetitions of the same problem has been exceeded, a new random problem is generated and presented to the robot.

The maximum number of repetitions of a problem has been set to 100. We must remember that noise makes possible that the repetition of one experience with the same policy could produce different results. Thus, the repetition of one failed experience could be successful without any change in the policy. The N parameter has been set to 100. The learning algorithm is used with all heuristics active. As we have discussed in section 6.3, the maximum number of steps and actions has been set to 2500 and 800 respectively. Finally, the learning process is stopped when, after 100 consecutive experiences, the policy is not changed.

We must note that this criterion for stopping the learning process does not ensure a totally reliable behavior, because 100 experiences are not enough to sample significantly all the different problems of this task. Unfortunately, to increase the number of experiences to stop the learning process is not useful because, in this case, the number of insolvable[4] problems that appear in that number of experiences is also increased. As not solved problems are repeated, the insolvable problems are repeated a lot of times. This normally results in a useless change of the policy. These experiences slow the learning process in two ways: time resources are spent in trying to solve these problems and, usually, the halting of the learning process has to wait for another cycle of 100 experiences. Thus, an equilibrium between the criterion for stopping the learning procedure and the number of experiences wasted for learning, must be found.

[4] Due to the low precision of the sensory system, the robot can found highly misleading states that cannot be solved. For instance, when the robot finds the free end of a brick, depending on the relative position of the robot, the robot cannot distinguish between a forward free state and a blocked one. It seems also that there is no procedure to disambiguate this state with the perceptual system while also solving the task. Fortunately, this states rarely appear (about 2% of cases).

Number of experiences

Robot	*Task*	Best	Worst	Average	Median
`R1`	`Going out of a room`	69	5778	1852.8	**964**

Reliability

Robot	*Task*	Best	Worst	Average	Median
`R1`	`Going out of a room`	99%	53%	85.1%	**88%**

Table 6.2: First results with robot `R1` using CANDID and `Active Memorization` with one memory bit. The number of experiences needed to learn the behavior and the percentage of reliability of the learned behaviors are shown for 30 independent runs.

We will see in the first experiments that results obtained with this stopping criterion are not totally reliable. Nevertheless, in order to solve this problem we will propose some solutions.

Results with 3 three-valued virtual sensors

The first experiment has been made with robot `R1` using CANDID and `Active Memorization` with one memory bit.

Results were obtained after the execution of 30 independent runs of the learning algorithm. The learning algorithm was executed with all heuristics active. The parameters for learning were 2500 and 800 as the maximum number of steps and actions (respectively) allowed per trial, 100 maximum repetitions of a failed experience, halting after 100 experiences without a change in the policy, and the N parameter set to 100. These parameters have been set without a search for the best results.

Results are presented in table 6.2. In the first table the *average* and *median* number of experiences needed to learn the behavior, are shown. Also the *best* and *worst* cases in 30 runs for this number of experiences are presented. In the second table, statistics about the reliability of the behaviors found in 30 runs are shown. There, the *average* and *median* percentage of problems solved for the 30 behaviors found are shown. Finally, the *best* and *worst* reliability found in the 30 runs are also presented.

The reliability for each of the 30 learned behaviors is estimated by keeping the number of successful experiences in a test performed when the learning process has been halted for 100 randomly chosen experiences. A random experience consists in choosing randomly a room and a location and orientation inside it.

As the reliability has been estimated by performing 100 randomly chosen experiences, if we consider the process as a *binomial statistical distribution* with 100 experiences, the result of each test can be considered with a maximum of $\pm 4\%$ error estimation with 95% of confidence. Nevertheless, we have performed 30 runs of the learning process. This reduced the interval estimation for the *median* and *average* results within a maximum of $\pm 1\%$ of error estimation (95% of confidence). *These values for the error estimation of reliability are valid for all the experiments described in this chapter, because in all of them we test each behavior with 100 random experiences and we execute 30 times the learning process.*

Results with 6 two-valued sensors

In this experiment we use the robot `R2` described in section 6.5. This robot uses all the sensors directly without grouping the information into a virtual sensor. In theory, if robot `R1` is able to learn the task, then robot `R2` also must be able to solve the task. To do that, the robot will have to learn the grouping function and the behavior at the same time.

Results has been obtained after the execution of 30 independent runs of the learning algorithm. The parameters used for learning are the same ones that were used with robot `R1`, that is, all heuristics used, 2500 and 800 as the maximum number of steps and actions (respectively) allowed per trial, 100 maximum repetitions of a failed experience, 100 experiences without a change in the policy to halt the learning process, and the N parameter set to 100. These parameters have been set without a previous search for the best results.

Results are shown in table 6.3. The first table displays the *average* and *median* number of experiences performed until the robot finds a no changing policy in 100 experiences. The *best* and *worst* cases in 30 runs for this number of experiences are also shown. The second table shows some statistics about the reliability of the behaviors found. There are presented the *average* and *median* percentage of problems solved for the 30 behaviors found, and the *best* and *worst* reliability found in the 30 runs.

A new information is also shown in the second table (that was not present in results for robot `R1`): *percentage of failures*. It expresses the percentage of runs that did not stop the learning process *before 6000 experiences* were performed. This limit has been set because we think that 6000 experiences is an excessive large number of experiences to be spent in a learning process for robotic tasks. Of course, this limit is debatable and it could be set higher or lower. We will indicate that a run has exceeded 6000 experiences with the 6000+ symbol.

Finally, notice that in the second table the average number of experiences needed to learn the task is not shown. We do not know the real number of experiences only for 35% of the runs (we only know this number for the runs that needed less than 6000 experiences) and thus, the average cannot be calculated.

Number of experiences

Robot	*Task*	Best	Worst	Average	Median	Failures
`R2`	`Going out of a room`	705	6000+	–	**3710**	**35**%

Reliability

Robot	*Task*	Best	Worst	Average	Median
`R2`	`Going out of a room`	90%	14%	61.9%	**76**%

Table 6.3: First results with robot `R2` using CANDID and `Active Memorization` with one memory bit. The number of experiences needed to learn the behavior and the percentage of reliability of the learned behaviors are shown for 30 independent runs.

Discussion of results

The best behaviors found by the two robots are completely described in appendix E. There, for each perception and value of the memory bit, the action to be executed is shown. Some instances of the execution of one learned behavior for the three different rooms can be observed in figures 6.4, 6.5 and 6.6. In these figures the complete trace of the robot for solving the task is depicted.

The performance of the obtained behaviors is similar to the one predicted in section 6.3: the robot searches for a wall, follows it and finally, when the exit door is detected, it goes out of the room.

In a careful study of these figures, we can notice the presence of noise in the environment. For instance, while the robot is following a wall the distance between the robot and the wall varies, due to the noise present in the motors that alters the robot's direction. This implies that sometimes the robot could loose contact with the wall. When this happens, the robot must be able to reestablish the contact. The behaviors learned with CANDID are able to do that.

In all the learned behaviors, the memorization action was used. Moreover, 30 runs with the same robots but without `Active Memorization` were not able to find an acceptable policy in less than 6000 experiences. *This shows that* `Active Memorization` *was needed and that it was successfully applied for this task.*

To analyze the results, we will study two main variables that describe the performance of the approach: the *reliability* of the learned behaviors and the *number of experiences* needed to learn the task.

The learned behaviors, unlike the ones obtained in the experiments described

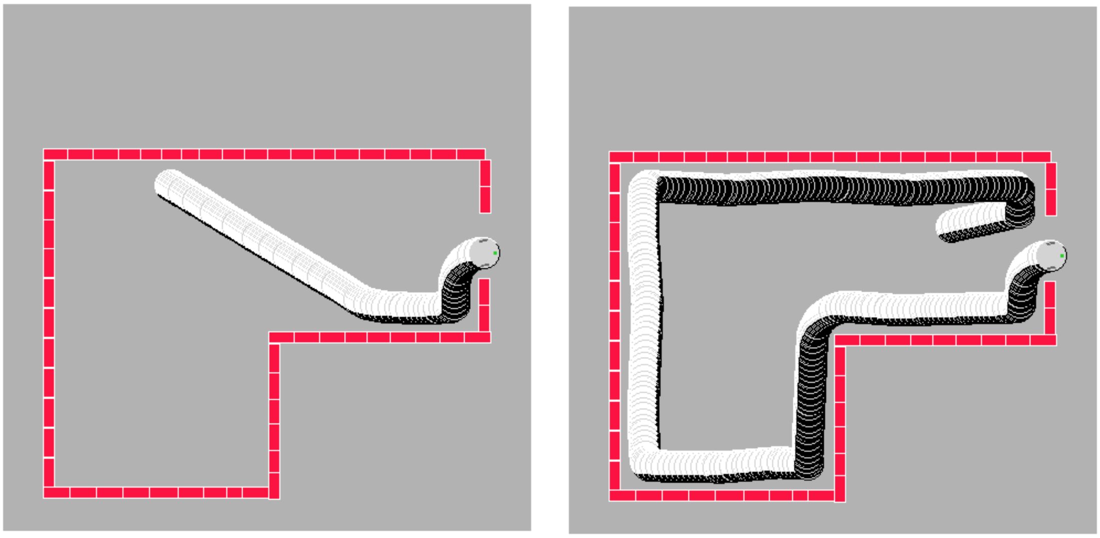


Figure 6.4: Trace of a learned behavior in the first room.

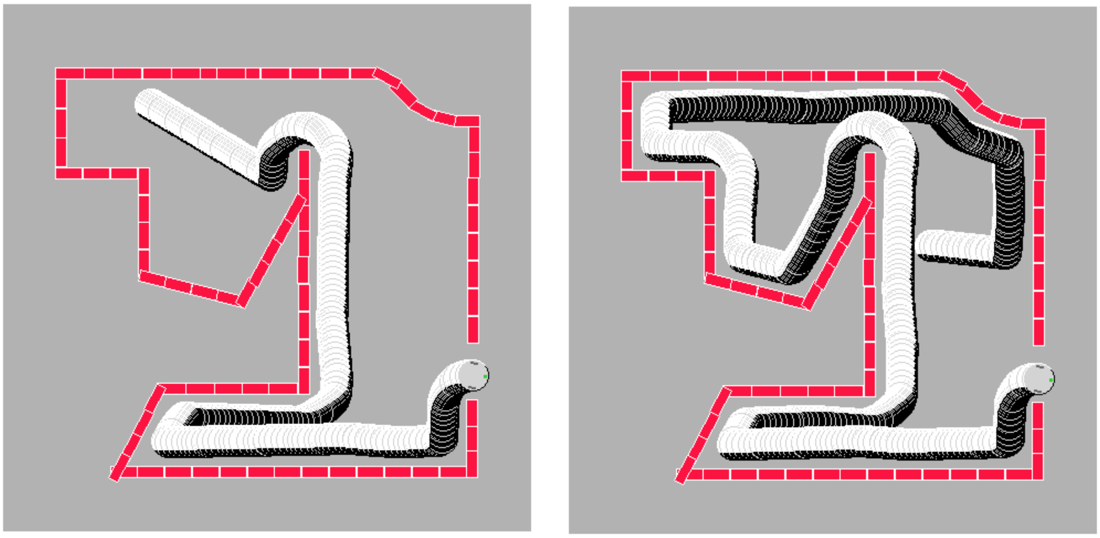


Figure 6.5: Trace of a learned behavior in the second room.

in the previous chapters, are not totally reliable. The reliability statistics of robot `R2` are not very high because in some runs it failed to learn the behavior. However, this explanation is not valid for the case of robot `R1` because the learning process stopped in all runs before the limit of 6000 experiences. A careful study of this case shows three main causes for the low reliability of the learned behaviors: a premature end of the learning process, some insolvable states, and noise.

A premature end of the learning process can be produced by a too low value in the number-of-experiences-without-changing-the-policy parameter used in the criterion to halt the learning process. This parameter has been set to 100 in this

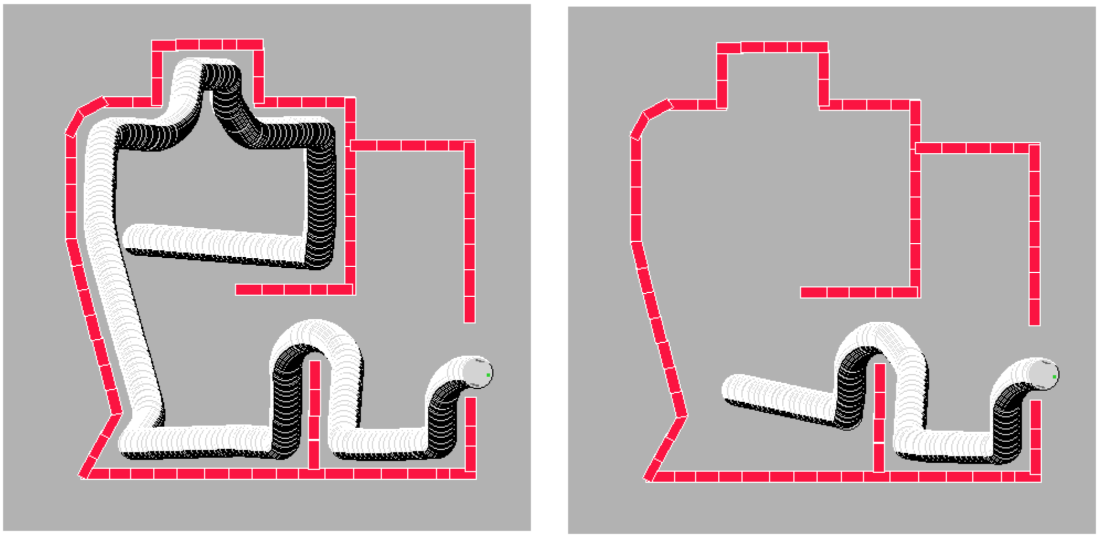


Figure 6.6: Trace of a learned behavior in the third room.

experiment, and it is not enough to test a significative part of initial states the robot can face (see section 3.3 for a discussion about the right value of the parameter and the consequences of a bad choice). Nevertheless, as we have discussed in section 6.5.1, this value cannot be increased if we want an efficient learning in the number of experiences.

The less reliable behavior obtained by `R1` is a typical example of this case: the learning process halted but a totally reliable policy has not been obtained. A further study of the performance of this behavior shows that it failed mainly in *some specific situations of one specific room* that the agent faces quite frequently. Nevertheless, bad luck in the random generation of problems caused that this kind of problems were not presented to the robot in the last 100 experiences. On the contrary, the problems generated were so easy to solve with the current behavior that did not change the policy, and thus, the learned process halted. Fortunately, a few additional training experiences (360) for these specific cases improved significantly the learned behavior, that resulted in 91% of reliability (above the average and median results). We will take this result as the basis of one mechanism for improving the reliability of behaviors (see section 6.7).

The second cause for a bad reliability is due to the presence of states that cannot be solved by the robot. These states have been described in the footnote of section 6.5.1. Although these states cannot be solved in the simulator, we are not sure that they could not be solved with the *real* robot if we allow some few collisions of the robot with the wall. We must remark that collisions of the robot with the wall are not dangerous for the robot.

Finally, another cause for failures is noise. Noise produces sometimes bad measures of sensors that fool the robot. In those cases, the robot executes an inadequate action for the real state and, sometimes, it collides with the wall producing a failed experience.

The second issue to be discussed about the results obtained in this experiment is the number of experiences spent in learning the behavior.

In the case of the robot `R1`, behaviors have been learned in an acceptable median number of experiences. Nevertheless, we must note that there is a considerable difference between the worst case and the median case, which indicates that the performance of the algorithm is not very stable for this task. This also can be observed because the average and median results also differ significantly. This feature is not desirable and must be corrected.

In the case of the second robot `R2` the median number of experiences is a lot larger than the ones required by robot `R1`. Moreover, 35% of the runs were not able to learn the behavior in less than 6000 experiences. The number of experiences needed to learn the behavior has to be somehow reduced for `R2`. In order to reduce this number of experiences, we propose an incremental approach that is discussed in the next section.

6.6 Incremental learning

The previous experiment has shown that CANDID is useful for learning in robotic tasks. Nevertheless, one important feature that must be improved is the high number of experiences required to learn the task.

To sped-up the learning of the task, we propose to use an incremental approach. In chapter 5 we proposed a new approach called `hierarchical development` that was successfully tested in the fruitcake problem. Although this mechanism could be also successfully applied to the current task, *we will propose in this section a new incremental approach* that combines memory with incremental learning.

The practical application of this new approach to the current task is derived from the observation of the learning process of the robots. We have noticed that most part of the initial failures of the robots was due to a bad dealing with the corners of the rooms. We think that learning could be accelerated if the robot first learns to *solve the corners of the rooms* and later (with this knowledge) it learns to solve the *going out of a room* task.

6.6.1 A new incremental approach

In this section, a new incremental procedure that combines incremental learning with memory, is described.

This incremental approach requires, first of all, a hierarchical decomposition of the complex task in subtasks. When this decomposition is made, the sub-tasks are learned with CANDID in the usual way.

Once the sub-behaviors are learned, in order to learn the original behavior, a memory bit with so many values as the number of subtasks learned plus one, is defined. Then, the behavior for the original task is initialized by storing in the table that will maintain the statistics for the behavior all the learned sub-behaviors in the following way: sub-behavior number i is stored in the zone of the table where the memory bit is set to i. In this way, the table in zones where the memory bit has assigned values from 1 to n (where n is the number of sub-tasks), will be occupied with the n learned sub-behaviors, while the zone of the table with memory bit set to 0 will be used to store the statistics that will define the original behavior.

With this initialization of the table for the higher behavior, the learning algorithm will be executed for the higher task in the usual way (remember to add the memory action for the new memory register with n possible values[5]). The agent can use the previously learned behaviors by executing the memory action that switches the control to the zone of the table where the desired behavior is defined. When the goal of the sub-behavior is achieved, the memory bit is automatically set to 0 (that is, the control is returned to the higher behavior). Updates are made in the usual way and, thus, the used sub-behaviors are tuned at the same time that the higher task is learned.

This hierarchical mechanism that we propose, has been derived from the close relation between hierarchical learning and memorization, that has been commented in section 4.2.6. This approach can be characterized with the parameters used in section 5.3.1 to analyze hierarchical approaches. From this point of view, this new mechanism does not need any kind of coordination, but builds a new behavior (as in Asada's work [9]). Learning is done in a stepwise way. However, at each step the previously learned behaviors are also updated (tuning of sub-behaviors). Finally, a reinforcement function for each sub-behavior must be defined, and the space of actions and perceptions has to be the same in all behaviors.

We must note that all conditions required to apply this new approach to our robotic task are fulfilled. Its application to the current task will be done in the following way.

First the robot is trained in the *solve the corners* task, without memory registers. This task, as we will define below, does not require memory to be solved. When this task has been learned, a new two-valued memory register (two values: one plus the number of learned sub-behaviors) and the corresponding memory action available are defined for learning the task of *going out of the room*. The behavior learned to *solve the corners* is stored in the zone of the table for the higher task where the memory bit is set to 1. Once the initialization is finished, the robot is trained in the the task of *going out of the room*. At each experience,

[5]Sometimes it will be better to use n memory actions. In this way, memory action number j will set the memory register to j.

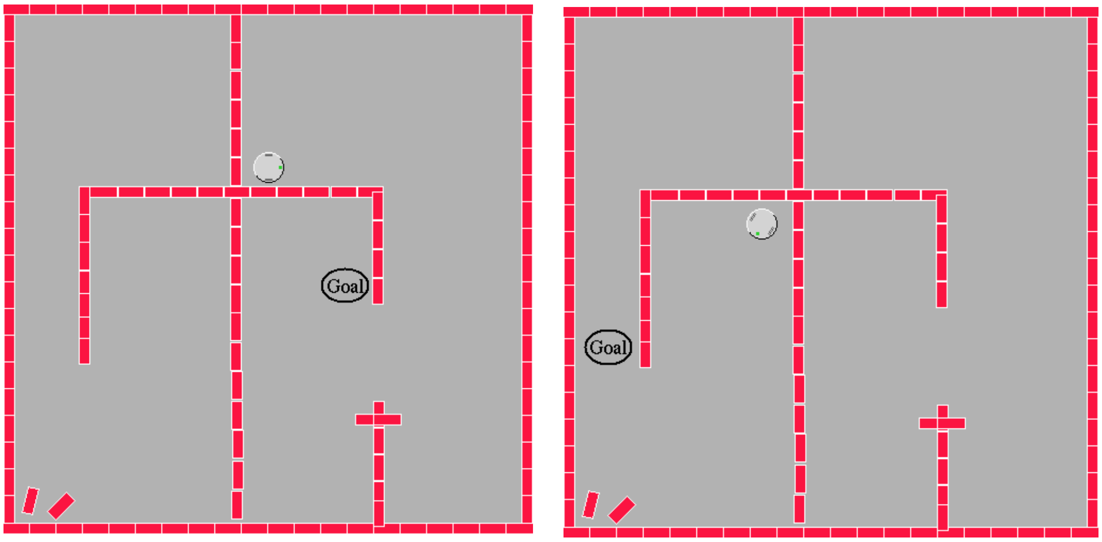


Figure 6.7: Two initial conditions and goal locations for each of them in the *solve corners* task.

the memory bit is initially set to 0. The robot can use the *solve the corners* behavior by executing the memory action. In this way, the agent switches the control to the zone of the table where the *solve the corners* task is defined.

6.6.2 Description of the *solve corners task*

The goal of this task is to successfully deal with the corners of the rooms while the robot follows a wall: the robot must maintain the contact with the wall while goes forward, avoiding collisions with the wall.

In order to learn this task, the robot has been trained in a specially designed *arena* where two kinds of corners that the robot could face in a typical room are represented: a closed corner and a open corner. The behavior to execute is different for each type of corner (although they can be learned in a single table). For each type of corner a different initial condition is defined. They can be observed in figure 6.7. The left snapshot shows the initial conditions for learning the open corner. The right snapshot shows the initial conditions to solve the closed corner. The two snapshots also show the goal location for each type of corner. The two corners are separated with a wall in order to prevent any interaction between the solving of the two different corners. In both cases, the robot starts in a location where the wall can be perceived, because we are only teaching the robot to *follow* the wall until it finds the goal location. Finally, in the bottom of the snapshots some bricks forming obstacles can be seen. They are there in order to prevent solutions to the task that do not follow the wall.

The robot is rewarded in this task when it achieves the goal location. The maximum number allowed of steps and actions per trial has been set to 400 and 100 respectively. The N parameter of CANDID has been set to 100, and the learning process is stopped when, after 100 consecutive experiences, the policy is not changed. The learning algorithm is executed with all heuristics active. These parameters have been set without a previous search for the best values. As we stated above, `Active Memorization` is not used to learn this task because it can be learned without memory bits.

Unlike previous experiments with CANDID, in this task there are only two initial situations. Thus, it is not necessary to generate random situations. In this case, the two different initial conditions will be presented alternatively to the robot. The maximum number of repetitions has been set to 10. A bigger number of repetitions is not adequate for this task because we are learning two different tasks in one table. In order to avoid interferences in the learning of the two tasks, when one task is learned (to avoid its unlearning), it must be often presented to the robot. In this way, in one behavior, the robot learns to solve the two kind of corners.

6.6.3 Results of incremental learning for robots `R1` and `R2`

Results with 3 three-valued virtual sensors

Results obtained by robot `R1`, using the new incremental approach proposed, are resumed in table 6.4.

These tables were obtained performing 30 runs of the incremental approach in the way described above. The parameters for learning the first task have been defined in section 6.6.2, and the parameters for the second task are the same that were used in section 6.5.1.

The first table shows, for each task, the *average* and *median* number of experiences performed until the robot finds the no changing policy in 100 experiences. Also, the *best* and *worst* cases in 30 runs for this number of experiences, are shown. Finally, the results obtained for these values *considering the whole process*, are shown under the title `Total task`.

The second table describes the reliability of the behaviors found in 30 runs. It shows the *average* and *median* percentage of problems solved for the 30 learned behaviors. Finally, it also presents the *best* and *worst* reliability results found in the 30 runs.

Comparing these results with the ones described in the previous section for robot `R1` (see table 6.2), we can notice that the improvement in the median value of the number of experiences needed to learn the task is moderated. However, the average and worst number of experiences are significantly improved.

We can also notice that the average and mean values are mutually closer with the incremental approach than in the previous case. This means that incremental

Number of experiences

Robot	*Task*	Best	Worst	Average	Median
`R1`	`Solve corners`	3	682	210	148
`R1`	`Going out of a room`	232	3555	839.4	621
`R1`	`Total`	309	3933	1049.5	**874**

Reliability

Robot	*Task*	Best	Worst	Average	Median
`R1`	`Going out of a room`	100%	64%	86.8%	**89**%

Table 6.4: Results with robot `R1` using CANDID, `Active Memorization` with one memory bit, and the *new* incremental approach. The number of experiences needed to learn the behavior are shown for the intermediate task, for the original task, and finally, for the sum of the two tasks. Statistics about the reliability of the learned behaviors for the original task in 30 independent runs are also shown.

learning is more stable than learning from scratch. Finally, reliability is slightly improved.

We conclude that, while reliability is not significantly improved in this case, *the learning process has been made more efficient and stable in the number of experiences.*

Results with 6 two-valued sensors

Results obtained by robot `R2` using the new incremental approach, are resumed in table 6.5.

As in the previous experiment, the tables were obtained performing 30 runs of the incremental approach. The parameters for learning are the same ones that were used with robot `R1`.

The first table shows, for each task, the *average* and *median* number of experiences performed until the robot finds the no changing policy in 100 experiences. Also, the *best* and *worst* number of experiences required in 30 runs are shown. Finally, the results obtained for these values *considering the whole process*, are also shown under the title `Total task`.

The second table describes the reliability of the behaviors found in 30 runs. It shows the *average* and *median* percentage of problems solved for the 30 learned behaviors. It also presents the *best* and *worst* reliability results found in the 30 runs.

Number of experiences

Robot	*Task*	Best	Worst	Average	Median
R2	Solve corners	7	4169	587.1	204
R2	Going out of a room	278	4607	1696.7	1250
R2	Total	467	5607	2283.8	**1333**

Reliability

Robot	*Task*	Best	Worst	Average	Median
R2	Going out of a room	92%	67%	81%	**80**%

Table 6.5: Results with robot R2 using CANDID, Active Memorization with one memory bit, and the *new* incremental approach. The number of experiences needed to learn the behavior are shown for the intermediate task, for the original task, and finally, for the sum of the two tasks. Statistics about the reliability of the learned behaviors for the original task in 30 independent runs are also shown.

Comparing these results with the ones obtained in the previous section for robot R2 (see table 6.3), we can notice that the improvement in the median value of the number of experiences needed to learn the task is very impressive: from 3710 experiences to only 1333. Moreover, in the incremental approach the learning process always stopped before 6000 experiences. On the other hand, median reliability of the learned behaviors is improved form 76% to 80%.

We can conclude in this case that the incremental approach has been useful to speed up the learning process and to increase the reliability of the learned behaviors. We must remember that the improvement in the performance of robot R2 was the main motivation for developing this new incremental approach.

6.7 Improving reliability: Extra-learning

The goal of reducing the number of experiences needed to stop the learning process has been achieved with the incremental approach described above. The other feature that remains to be improved is the *reliability* of the learned behaviors.

The reliability of the learned behaviors with CANDID mainly depends on the criterion for stopping the learning process. It must be stopped when the current behavior is totally reliable. The criterion used by CANDID to determine this circumstance has been described in section 3.3. It consists in halting the learning process when the current policy has not been changed after performing a given number of experiences. This number of experiences is a parameter that must be given by the user.

In the discussion of results in section 6.5.1, the application of this criterion to the current task is discussed. The parameter for stopping the learning process cannot be increased because the influence of no solvable problems delays too much its ending. However, the trouble with a low number of experiences that determines the stopping of the learning process is that the probability of a premature end of the learning process is quite high.

Nevertheless, in the same section, we mentioned that when the learning process has stopped prematurely, the learning of supplementary experiences becomes useful to increase the reliability of the learned behavior. We will implement this idea in order to improve the reliability of learned behaviors: the usual *learning process will be continued until another time the stopping criterion is fulfilled* (that is, in our case, until another time 100 experiences have been performed without changing the policy). Obviously, this procedure can be generalized to continue the learning until a parameterized number of times the stopping criterion is fulfilled. We call this extension of the learning process `extra-learning`.

Experiments show that this procedure is better than increasing the parameter used to stop the learning process in both the reliability of the learned behaviors and the number of experiences required to learn the behavior. For instance, to stop the learning process when the criterion for 100 experiences is fulfilled two times, shows better results than stopping when the criterion is fulfilled one time for 200 experiences.

Results of extra-learning

A set of 30 runs of the `extra-learning` procedure has been performed for the behaviors obtained with robot `R1` in section 6.5.1.

Table 6.6 shows statistics about the number of experiences spent in the learning process for the conventional learning (under the title `First learning`) and for the extension of the learning process (under the title `Extra-learning`). These results are resumed *considering the whole process* in the `Total` row. The other table presents some statistics about the reliability of the behaviors found at the end of the whole learning process.

These results show that the obtained increase in reliability is statistically relevant (about 5% when the maximum improvement possible is 13%). Nevertheless, it has an added cost: the number of experiences spent in the `extra-learning` process. About 500 additional experiences are needed to improve in this way the reliability of behaviors.

Number of experiences

Phase	*Robot*	*Task*	Best	Worst	Average	Median
`First learning`	`R1`	`Going out of a room`	69	6778	1852.8	964
`Extra − learning`	`R1`	`Going out of a room`	29	3249	559.5	281
`Total`	`R1`	`Going out of a room`	460	7056	2412.3	**1589**

Reliability

Robot	*Task*	Best	Worst	Average	Median
`R1`	`Going out of a room`	100%	71%	90.2%	**92**%

Table 6.6: Results obtained with robot `R1` using CANDID, `Active Memorization` with one memory bit. There are shown the number of experiences needed to learn the behavior in the common way (the results shown in table 6.2), the number of experiences needed in the extra-learning, and finally, the accumulated number of experiences for the two phases. Statistics about the reliability of the learned behaviors for the original task in 30 independent runs are also shown.

Results allowing some collisions

The behaviors obtained, although improved with the `extra-learning` procedure, are not so reliable as desired. We have mentioned in section 6.5.1 that because of the limited perceptual system of the robot, there are 2% of not solvable problems in this task. This puts a limit on the maximum reliability possible to obtain at 98%.

We have seen also in the same section that one possible cause for failed experiences is noise. Noise produces sometimes bad measures of sensors that can fool the robot. In those cases, the robot executes an inadequate action for the true state and, sometimes, it collides with the wall yielding a failed experience.

In order to measure this effect, we will relax the conditions for a failure *in the test* of behaviors. These collisions are not a danger to the robot. We do not allow them in the learning phase because we do not want to obtain behaviors that *usually* collides with the walls in order to solve the task.

A re-evaluation of the behaviors learned with robot `R1` after `extra-learning`, but now allowing collisions, give us the following results: the average reliability increases from 90.2 to 92.4% and the median reliability increases from 92% to **95%**. In those cases, the average number of collisions in a successful experience is less than 0.1% of the steps performed by the robot, while the median number of collisions in a successful experience is 0 (that is, no collisions in the common

Number of experiences

Robot	*Task*	Best	Worst	Average	Median	Failures
`R1`	`Solve corners`	73	1191	622.6	670	0%
`R1`	`Going out of a room`	3338	6000+	–	6000+	70%
`R1`	`Total`	4257	6000+	–	**6000+**	**70**%

Reliability

Robot	*Task*	Best	Worst	Average	Median
`R1`	`Going out of a room`	92%	4%	67.9%	**83**%

Table 6.7: Number of experiences needed for learning the behavior in the incremental way and percentage of reliability of the learned behaviors for 30 independent runs *without noise*. Compare these results with the ones obtained with noise in table 6.4.

experiences).

These results show that the final reliability obtained with Candid *is very high.* The 5% of experiences in the median case that remain to be solved are due to the no solvable problems in the simulator (2%) and to the premature end of the learning process in some of the 30 runs.

6.8 The role of noise

In this section, we study how noise affects the performance of the learning process in this task. In order to make this analysis, we will inhibit noise in the simulator. This modification has not been made to improve the results obtained with Candid, but only to discover to what extent noise influences the performance of the learning algorithm in this task.

Surprisingly, results show that presence of noise not only is not so bad as we could think *a priori*, but that it speeds-up the learning process!

Results for 30 runs of Candid with `R1` using the incremental approach *with noise inhibited*, are shown in the table 6.7.

The experiments have been executed with the same parameters that were used in the incremental experiments with `R1` and noise (see section 6.6.3). Surprisingly, the algorithm failed to stop the learning process before performing 6000 experiences in 70% of the runs, and the minimum number of experiences needed was

3338. *Clearly, the results in the number of experiences needed to learn the behavior are a lot higher in this case than in the same conditions but without noise* (compare table 6.7 with table 6.4).

With respect to the reliability of the behaviors obtained, the results obtained with noise are also better than the ones obtained without noise. The best policy obtained without noise is 92% reliable, while the median policy is 83% reliable.

In order to clarify this surprising results, we have made another experiment. It consisted in testing the policies obtained with noise in the same task *but now with noise inhibited.* A priori, the test without noise of behaviors learned with noise, should return better reliability values than the values obtained testing the same behaviors with noise but, surprisingly, *results show that the reliability of these behaviors decreases when they are tested without noise!* For instance, the best policy learned with noise, that was 100% reliable in a test of 100 experiences (it means that it is above the 96% of reliability with a confidence above the 95%), returned a reliability of 81% in a test of 100 experiences without noise (it means that it is below 85% of reliability with a probability of 95%)! *These results make sense only if the learned behaviors use noise to solve the task, that is, the behaviors learned with noise also learn to take advantage of noise to solve the task.*

In conclusion, when the robot learns without noise, it requires a lot more experiences to learn than learning without noise. Moreover, behaviors learned with noise show a worse reliability when they are tested without noise. The explanation of these surprising results is not easy: it seems that CANDID can take advantage of noise in this task, but which are the concrete mechanisms that allow these results? We do not have a satisfactory answer to this question. Nevertheless, we have noted that policies learned and working with noise show in some locations of the rooms (for instance in its corners) a doubtful conduct, choosing one action and later the opposed one. In these cases, the robot seems to remain blocked in that location during a few moments until it takes the control again and goes out of that location without bumping into the wall. When the same policies are tested without noise, the robot is not able to go out of these locations and remains there until the experience is failed because the maximum number of steps allowed is exceeded. This seems to indicate that the robot has really learned policies that takes advantage of noise.

However, there remain more questions that must be answered. For instance, why this task without noise is not learned in 70% of the runs? This question is not easy to answer, either. On one hand, as we have seen in the characterization of the task in section 6.5, the task to be learned presents a lot higher number of possible policies than any of the tasks presented in the previous chapters. Thus, it seems reasonable to think that this task would require a larger number of experiences to learn a reliable behavior. In addition, the absence of noise reduces the number of acceptable policies to only those that do not use noise to solve the task (we have seen above that these policies do exist). These two factors could explain in part the high number of failures.

Another cause for the delay in the end of the learning process could be related to the following question: why the algorithm failed to learn a reliable behavior in 70% of the runs while the mean reliability achieved is so high (80%)? The absence of noise makes that the repetition of a failed experience with the same behavior, always produces the same results. The different effects of the repetition of failed experiences (aimed to obtain totally reliable behaviors) when it is used with noise compared with when it is used without noise, could explain also in part the difference in the number of experiences needed to stop the learning process.

We must note that this results should be studied in more detail, because they show an *a priori* counter-intuitive result. Although some preliminary explanations have been proposed, the results could hide more important reasons for this surprising performance. One possible explanation of these results could be that noise allows a subtle kind of generalization in the learning process that speeds it up. If this intuition is someday demonstrated, this generalization procedure could be applied in reinforcement learning mechanisms to accelerate the learning process.

6.9 Conclusions

The aim of this chapter was to show that *the learning mechanisms proposed in this thesis can be successfully applied to robotic tasks that present some realistic features (as noise and a continuous environment)* that were not present in previous experiments.

In order to achieve this goal, we have selected a realistic robot simulator that presents these features (the `Khepera simulator`; a simulation software of the `Khepera` robot). The task designed to test the capabilities of Candid and `Active Memorization` in this simulator was *going out of any room.*

First experiences using Candid and `Active Memorization` show that they are able to produce highly reliable behaviors for the selected task in the simulator. It is interesting to note that although no modifications were introduced in the Candid algorithm for facing noise or a continuous environment, it was able to deal with them. Nevertheless, the obtained results were not so good as it was expected. On one hand, experiments showed that in some cases the number of experiences needed to learn the behaviors was too large for a robotic task. On the other hand, the learned behaviors were not totally reliable. In order to improve these results, two new mechanisms have been proposed.

On one hand, in order to reduce the number of experiences needed to learn the task, we have proposed a *new incremental approach* that combines memory with hierarchical learning. The experiences performed with this new approach show that it actually reduces the number of experiences spent for learning the task.

Robot	*Method*	*Nickname*	N. exps.	Reliability	Failures
`R1`	CANDID and `Active Mem.`	A	964	88%	0%
`R1`	A + `Incremental`	B	874	89%	0%
`R1`	A with `Extra − learning`	C	1589	92%	0%
`R1`	C with collisions allowed in test	D	1589	95%	0%
`R2`	CANDID and `Active Mem.`	E	3710	76%	35%
`R2`	E + `Incremental`	F	1333	80%	0%
`R1`	A without noise	G	6000+	83%	70%

Table 6.8: Summary of the experiments performed in this chapter.

On the other hand, in order to improve the reliability of the learned behaviors, we have proposed an extension of the learning process that we call `extra-learning`. This extension of the learning process has been also successfully tested in the proposed task.

Table 6.8 shows a summary of the results obtained with the mechanisms proposed in this chapter. In this table, it can be observed the increase of reliability obtained using the `extra-learning` procedure and the decrease of the number of experiences spent in the learning process produced by using the new incremental mechanism. Final results show a high reliability of the learned behaviors in a moderated number of experiences. The final reliability obtained with CANDID is very high (95%). The 5% of experiences in the median case that remain to be solved could be due to the no solvable problems with the simulator (2%) and to the premature end of the learning in some of the 30 runs.

Thus, we have shown that the learning mechanisms proposed in this work can be fruitfully applied to robotic tasks, fulfilling the required constraints about efficiency, continuous environment and noise.

Instances of the learned behavior for the proposed task are available in some movies at the `http://www-lsi.upc.es/~mmartin/tesis.html` Web address.

Chapter 7

Conclusions

7.1 Contributions

Our main motivation in this work was to develop mechanisms for scaling up reinforcement learning algorithms enough to learn complex tasks for embedded agents. To achieve this goal, we have developed a set of algorithms and mechanisms throughout the chapters of this work.

In **chapter 2**, we have done an *analysis of the features that are present in complex tasks*. From this analysis, we have identified and discussed those features that make more difficult the learning of a task by embedded agents (section 2.2). The other ones are also discussed but as features subordinated to the relevant ones (section 2.5).

Through the analysis of these features, we outlined *a new approach* that has been developed in this thesis (section 2.3). This new proposal is composed of a *new reinforcement learning algorithm* designed to learn general and reliable behaviors, a *new mechanism to solve the perceptual aliasing problem*, and a *hierarchical methodology that allows incremental learning of complex behaviors* from previously learned ones.

Finally, the features that a reinforcement learning algorithm has to show in order to be fitted to our approach, are described (section 2.4).

In **chapter 3**, we have presented *a new learning algorithm, called* Candid (see section 3.1), that fulfills the properties required by the approach outlined in chapter 2.

The new algorithm implies a change of focus in the usual reinforcement learning algorithms, *because it has been designed to learn general and reliable behaviors.* We think that this change of focus is necessary in order to tackle complex tasks.

In order to develop this approach, an analysis of the conditions required for learning general behaviors has been done. The first consequence of the analysis is *the discovery that for learning general behaviors with limited perception, the* *classical concept of optimal behavior is not useful* (see section 3.5.1). In this case, we propose a *change from searching for optimal behaviors to searching for acceptable behaviors.*

Learning general behaviors is more difficult than learning specific behaviors, because the agent can use for learning only features that are present in all problems of the task. Sometimes, the algorithm finds a good policy for the most usual problems that is not totally reliable. In this case, it would fail for some rare problems. Nevertheless, the policy found is so good for the other problems, that the algorithm becomes reluctant to change it. *In order to force the algorithm to learn also these rare problems, and to return acceptable general and reliable behaviors, a new strategy that consists in the repetition of failed experiences, is proposed* (see section 3.2.2).

Another feature the learning algorithm fulfills is that it is robust to the perceptual aliasing problem (see section 3.5.1). CANDID has been designed to allow solving tasks even under a low degree of perceptual aliasing. It has been achieved by basing the learning procedure on the final reinforcement obtained. Although CANDID is not able to completely solve the perceptual aliasing problem (we have proposed in chapter 4 a complementary approach to solve this problem), the robustness of the algorithm allows to decrease the amount of resources necessary to solve it.

A new strategy, incorporated in the learning mechanism, allows to automatically determine when no improvement in the learning process is obtained. Then, the algorithm is able to automatically detect when the learning process for the general behavior has to be stopped. We show that this new procedure, based on the stability of the obtained policy, is better than other more naive procedures that determine the end of the learning process based on the number of failures in the last experiences. This feature is required for the new hierarchical learning mechanism proposed in chapter 5.

Another new feature of the learning algorithm is its weighting procedure that actuates, compared with other classical approaches to reinforcement learning, as a variable learning rate parameter α (see section 3.5.1). It produces a considerably acceleration of the learning process (see results in section 3.5.2). Some adaptations of this procedure could be applied to other classical reinforcement algorithms, for instance in `Q-learning`.

We have shown that all this new features of the learning algorithm become useful for learning general and reliable behaviors. CANDID has been successfully tested in a test task. Other known reinforcement algorithms have failed to solve this task or have solved the task at least two times slowly than CANDID (see section 3.5.2).

Other new contributions consist in variations of the CANDID *algorithm proposed to accelerate its learning process* (see section 3.4). Some of these variations could be successfully applied to classical reinforcement learning algorithms (for instance, to `TD` methods), specially the *variable punishment* variation and heuristics `H1` (only for trial-based tasks) and `H2`.

Heuristics `H3`, `H4` and `H5` are designed in order to make the learning process performed by CANDID more systematic. When they are used together with heuristics `H1` and `H2`, another interesting feature of CANDID appears: *when the whole set of heuristics is used, the algorithm does not redundantly explore policies previously tested unsuccessfully, at least, until a first positive reinforcement is obtained* (see section 3.4.2 and appendix C). This implies a high acceleration of the learning process performed by CANDID. Classical reinforcement learning algorithms, as `Q-learning` or `TD` methods, also tend to repeat the exploration of no successful states. Some smart exploration mechanisms have been proposed in order to avoid this. Unfortunately, these exploration mechanisms are not useful for tasks with perceptual aliasing (see section 5.1.3). The set of heuristics we propose, plays for CANDID the same role as these exploration mechanisms for classical reinforcement learning algorithms. The heuristics change the techniques that avoid to visit previously explored states unsuccessfully (techniques that are not useful for tasks with perceptual aliasing because these states cannot be discriminated) by techniques that avoid to test previously tested policies. *These heuristics are, then, a new contribution to reinforcement learning because they define the first mechanism (we are aware of) that is useful for tasks with perceptual aliasing for avoiding the repeated testing of failed policies.*

Finally, it is also interesting to note that CANDID needs less parameters for learning than other reinforcement algorithms, and that the values chosen for these parameters are easy to find and not critical (that is, a bad choice of the values of the parameters, usually does not prevent learning a behavior, but only slightly influences learning speed or the efficiency of the found behavior).

In **chapter 4**, a *comparative study of the different approaches used to solve the perceptual aliasing problem* is performed (see section 4.2).

From it, we propose a new strategy to solve the perceptual aliasing problem that is suited to embedded agents. The new proposal is called active memorization and consists in two different architectures, `Active Memorization I` *and* `Active Memorization II` *(see section 4.3.2), that must be used depending on the task the agent is facing.*

Experiments performed showed that these two architectures are appropriate to solve the perceptual aliasing problem, and that they solve more efficiently than other approaches, an extensive set of benchmark tasks (see results for these architectures in section 4.4).

The main difference between the active memorization approach and other memory approaches (as the window approach) consists in the different selective process used to store the information that allows to solve the perceptual aliasing problem. Our approach is much more selective than other approaches based on memory, for instance the window approach (see section 4.2.8) where all last t actions and/or situations experienced by the agent are automatically stored. These memory approaches (see section 4.2.8) become passive in the way that historical information is stored, because it is done automatically. In this way, relevant and not relevant information is stored and taken into account in order to solve the perceptual aliasing problem. When the amount of irrelevant information stored becomes too high, the learning of the task becomes too difficult.

On the contrary, our approach does not need to assume the continuum hypothesis (that all last t situations are relevant for solving the perceptual aliasing problem). *With active memorization, the agent learns not only how to solve the problem with external actions, but also how to actively collect the necessary information to disambiguate aliased states.* For instance, when the agent passes through an state and it would be useful (in order to solve the perceptual aliasing problem) to remember this, the agent must actively memorize it by executing a memory action.

Our approach is also different to previous attempts to solve the perceptual aliasing problem by using memory actions (for instance Littman's and ZCS *approaches; see section 4.3.1) because it does not show the exponential growth of actions that makes these previous attempts computationally intractable.*

Also, on the merits side, it is interesting to note that our approach can solve the perceptual aliasing problem without some constraints that are present in other methods that make no use of memory. For instance, it is possible to visit aliased states without any risk; this was not possible in the *ignoring* and the *avoiding aliased states* approaches (see sections 4.2.1 and 4.2.2). It is possible to learn a behavior without having a model of the environment which is suited to the view of an embedded agent that learns only interacting with the environment without any *a priori* knowledge (in the *belief vectors* approach a world model is necessary; see section 4.2.3). Finally, in our approach it is not necessary to learn a world model of the environment; in the *disambiguate states by learning a model* approach (see section 4.2.3), it is necessary.

We must remark that our approach, by being independent of a world model, allows learning general behaviors. Remember that this behaviors are useful for different environments. This is not the same as learning optimal solutions for particular problems. *Our approach is, then, an adequate extension to the* CANDID *algorithm.*

Finally, we must note that the active memorization approach works well in a extensive set of navigation environments and solves problems that include an extreme lack of perceptual information and tasks that include a large number of real states (see results in section 4.4). In some cases, the approach produced even better results than algorithms that use a complete information of the environment.

Chapter 5 was devoted to scale up our reinforcement learning approach to problems that show a long delayed reinforcement.

First, *an analysis of the existing approaches to scale up reinforcement learning to this kind of tasks* has been done (see section 5.1). In this analysis we have criticized each approach in terms of its possible application to embedded agents with a limited perceptual system for learning general behaviors.

From the analysis, *we have proposed and tested two different mechanisms to solve the problem of a long delayed reinforcement.* This two approaches are based on the incremental learning approach that proposes a stepwise learning process going from the simplest problems/tasks to the more complex problems/tasks.

The first approach is used for the *incremental learning of more complex problems of the same kind of task* (see section 5.2). This implies the incremental learning of the same behavior. First the easy problems of the task are presented to the agent. Once they are learned, more complex problems are proposed and so on until the agent has learned to solve the most complex problems of the task with a single behavior.

Experiences performed show that this mechanism can be successfully applied to the CANDID algorithm (see section 5.2). The behaviors obtained by using this incremental approach are more efficient and/or more quickly obtained than the behaviors found without using the hierarchical approach. Nevertheless, a more extensive study of this mechanism has to be done.

The second approach to incremental learning that we propose, is based in the hierarchical *learning of new complex tasks* by learning intermediate behaviors (see section 5.3). This kind of tasks presents a very long delayed reinforcement when the agent tries to solve them using only its initial set of actions, and (unlike the previous incremental approach) could not be learned by using only the CANDID algorithm. Using the learned intermediate behaviors as primitive actions, the original complex task can be solved in a few steps, circumventing then the problem of the long delayed reinforcement. *Our new proposal, called* `hierarchical development`*, allows this kind of hierarchical learning.*

In order to design this proposal, we have done a previous *analysis of the features that characterize hierarchical learning approaches* (see section 5.3.1), *searching for those ones that can make the approach more efficient. Our approach presents those features: delegatory coordination, different reinforcement functions for each sub-behavior, and separate spaces of actions and perceptions for each sub-behavior* (see section 5.4 for a description of the features, and see section 5.4 for knowing how are fulfilled these features in our approach).

Our new approach has been tested in a well known task: the *fruitcake problem.* This task was tackled in previous researches [34] [206], but it was only partially learned by them. *We have shown by using the* `hierarchical development` *approach that this task can be learned by embedded agents* (see section 5.5).

Another novel item that must be remarked in our approach is the tuning mechanism (see section 5.4). When the sub-behaviors used in a complex behavior are not totally reliable, the performance of the complex behavior is seriously affected (see the justification in section 2.4.3). In order to solve this problem, learning (tuning) of sub-behaviors is allowed when the behavior for the complex task does not improve its performance. *It is important to note that the tuning procedure is automatically enabled.* There is no need to indicate when the learning of sub-behaviors has to be begin. As we have seen in the "maximum number of experiences without a change in the policy" parameter in section 3.3, CANDID can detect when the performance of a behavior does not improve; when this happens, the tuning procedure is activated.

Finally, in **chapter 6** *we adapted the approach proposed in this work to robotic tasks*. By doing so, we showed that the approach is suited for tasks that present more realistic features as *noise* and a *continuous environment*. We used the `Khepera simulator`; a realistic simulation software of the `Khepera` robot that presents these features.

First experiences using CANDID and `Active Memorization` showed that they are able to produce highly reliable behaviors for the selected task in the simulator. *It is interesting to note that although no modifications were introduced in the* CANDID *algorithm for facing noise or a continuous environment, it was able to deal with them.* Nevertheless, results obtained were not so good as it was expected. On one hand, experiments showed that in some case the number of experiences needed to learn the behaviors was too large for a robotic tasks. On the other hand, the reliability of the learned behaviors was not totally satisfactory.

In order to speed up the learning process in the number of experiences needed to perform, we have proposed a new incremental approach that combines memory with hierarchical learning. The experiences performed with this new approach show that it actually reduces the number of experiences spent for learning the task.

In order to improve the reliability of the learned behaviors, we have proposed an extension of the learning process that we call `extra learning`. This extension of the learning process has been also successfully tested in the proposed task.

The results obtained with those improvements show that the learning mechanisms proposed in this thesis can be fruitfully applied to robotic tasks, fulfilling the required constraints of efficiency, continuous environments and noise.

7.2 Future work

This work is not closed. There are several challenging issues that can improve the approach presented in the thesis. We divide the future work depending in the following sections.

7.2.1 CANDID

One of the most interesting future works for CANDID is to extend it with an appropriate mechanism in order to allow the *generalization of perceptions* in the learning process.

Generalization consist in grouping, somehow, the agent's sensory inputs that could have the same action to execute in an acceptable behavior to solve the proposed task. The generalization procedure has two profitable effects; it reduces the size of the spatial resources used to build the behavior and, more important, it allows to speed the learning by suggesting for a new perceptual state the same response that another known *similar* state (that usually becomes more appropriate than a random guess).

The use of deictic sensors that report incomplete information about the environment implies, in fact, a forced generalization process. Several different real states perceived with a limited sensory system could be perceived as the same state. This implies a generalization of situations forced by the agent's perceptual system. Following this idea, some work has been done. Experiences has been carried out forcing a generalization of states by means of a limitation of the agent's perceptual system. This limitation can inhibit the possibility of learning to solve the task, but also can increase the speed of learning. Experiences has been performed in in order to empirically find the optimum balance between the two effects. We have performed this study applying random neural networks [186] and genetic algorithms [70] to the framework presented through this work. Some results have been published in [110], but a more deep study has to be done.

Another more standard way to include the generalization of perceptions in the learning process, is to use a neural network to implement the CANDID algorithm, in a similar way that `Q-learning` generalize states with `CMAC` neural networks [186] or with *feed-forward neural networks* trained with the `back-propagation` algorithm (for instance [91] or [158])

Another interesting extension of CANDID consists in the development of procedures able to *generate optimal solutions for a specific problem starting from the behavior obtained for the general task*. This extension can be seen as a procedure that refines general behaviors for obtaining optimal solutions for particular problems.

This could be an efficient way to find optimum solutions for similar particular problems. The base from which this procedure starts (the general behavior) ensures the solving of all the problems. Thus, it would be better, in order to find the optimum solutions for the particular problems, to start form the general behavior than to start from scratch.

Another future line of work for CANDID consists in developing an efficient procedure to allow the agent to *learn from examples of successful experiences* of the task. In this case, an example would consist of a successful sequence of actions executed to solve the current problem. The examples could be provided by a teacher or by monitoring the actions performed by other agents that know how to solve the task.

Finally, another interesting line of work is to extend the CANDID algorithm to the kind of tasks that are not defined as achieving a goal task, but as the permanent fulfilling of one condition, for instance the *survive* task is defined as the permanent fulfilling of the *be alive* condition.

7.2.2 Active Memorization

There are some open questions about the active memorization approach that we introduced for solving the perceptual aliasing problem, that could be interesting to study.

The first one is about the number of memory registers necessary for solving a task. This is a critical parameter that must be given by the teacher. It could be interesting, then, to develop a methodology that would allow an *a priori* determination of how many memory bits are necessary for a given task.

If this is not feasible, an alternative approach could be the design of a procedure that could determine (while the algorithm is running) when the used memory bits are insufficient to solve the perceptual aliasing problem. This procedure would be used to dynamically increase, accordingly, the memory resources of the agent.

Another interesting issue is to study whether the active memorization approach can be extended successfully to other algorithms or not, for instance `Q-learning`, in order to find optimal policies.

It is possible that this approach could not be successfully applied to classical reinforcement learning mechanisms. One of the reason that makes successful the application of active memorization to CANDID is that the algorithm does not use any exploration mechanism. It has been suggested in [208] that taking random actions periodically, could be incompatible with the adequate use of memory actions.

Nevertheless, there is no formal proof for this statement and it would be interesting to make a deep study of the possible application of the active memorization approach to classical reinforcement algorithms.

7.2.3 Hierarchical development

In the `hierarchical development` methodology proposed in chapter 5, the teacher must give the following information: the decomposition of the task in sub-tasks, and the spaces of actions and perceptions for each subtask.

Although it seems that human teachers can do it easily, in some cases (for instance, in totally autonomous agents or in *animats*) it would be preferable to obtain this information automatically. An interesting future work will be to develop a procedure able to automatically obtain this information. It seems that there exists some mechanisms that probably could do that.

One possible way to automatically find the decomposition of a task in sub-tasks and the set of actions and perceptions relevant for the each sub-task, is by using genetic algorithms [115]. In this case, the genetic algorithm would have to find an ordered set of reinforcement functions. Each reinforcement function describes a task to be learned, and the order in the set describes the sequence in which the tasks must be learned. With each of these reinforcement functions, the genetic algorithm also would have to propose a set of useful actions and perceptions to learn the sub-task.

All this information would define an individual of the population maintained by the genetic algorithm [70]. This is the necessary information to use the `hierarchical development` approach. Then, this approach would be applied for all individuals of the population, and the results would be evaluated with an adequate fitness function. From this evaluation, the genetic algorithm would generate the new population and the procedure would be repeated. Finally, the genetic algorithm would return, after several generations, one individual that maximizes the fitness function (that is, a list of sub-tasks and spaces of actions and perceptions for each sub-task, that maximizes the fitness function).

Nevertheless, this is not a trivial application of genetic algorithms because the global fitness function used to evaluate the proposals has to be designed in order not to produce a zero return in any of the failed executions of `hierarchical development`. The fitness function has to differentiate among failed executions, those ones that are closer to solve the complex task.