Master Thesis
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Reinforcement Learning on Self-reconfigurable Modular Robots

Learning as a methodology for solving locomotion and reconfiguration tasks

Student: Ricardo Martín Kirkner (L.U. 479/98)

Director: Dr. Juan Miguel Santos

March 1st, 2007

Abstract

Reinforcement Learning has been found repeatedly missing in the literature about reconfiguration and locomotion tasks in Self-Reconfigurable Modular Robots.

This methodology, however, has been proved to produce significant benefits in those cases where it was used to solve problems related to dynamic and unpredictable environments. This kind of environments is very common when dealing with Self-Reconfigurable Modular Robots.

In this work, we first survey the state of the art in the field of Self-Reconfigurable Modular Robots, in order to determine what the yet unsolved problems and the possible niches of research are. We then develop a simulator in order to study the Reinforcement Learning methodology applied to the generation of behaviours that efficiently solve reconfiguration and locomotion tasks using the M-TRAN Self-Reconfigurable Modular Robot. Finally, several experiments are suggested in order to test this methodology. For those experiments it is necessary to define the best representation for the state-action space, and the way to discretize it in order to make those experiments computationally feasible. Another issue analyzed is the impact of several aspects involved in Reinforcement Learning problems, such as the definition of an episode, the learning velocity and policy evaluation.

Since this kind of study involves a significant experimental component, after designing and performing the experiments, the results are analyzed and conclusions are then drawn about the effectiveness and possibilities of Reinforcement Learning when applied to these kind of robots.

Contents

Chapter 1

Introduction

We are immersed in a reality in which robots constitute an increasingly common and necessary element for solving tasks with higher efficiency and lower risk for humans. Robots are present everywhere, from industry to entertainment and medicine. However, in most cases, these robots are restricted to a well known and not very flexible environment. Should the environment change, the robot might be rendered partial or completely unusable, since it would stop complying to the conditions for which it was designed.

There are many types of robots, with different characteristics. They can be used as mobile platforms, as manipulators and even as mobile laboratories. Due to the fact that they operate in environments with rigid constraints, they cannot be used in scenarios for which they where not designed.

It is here where *Self-Reconfigurable Modular Robots* shine. These robots are composed of several independent modules, which can be combined in several ways. Thanks to this, it is possible to build different kinds of specialized robots from the same modules. These robots can also change their own structure so the better adapt to each individual task and environment in which they have to operate.

It is because of this that these type of robots can overcome the limitations of non-reconfigurable robots. Being modular, they can acquire different capacities on different occasions, and for example, combine the capacities of a mobile platform with those of a manipulator, so to obtain the benefits of both kind of robots.

On the other hand, in the field of Machine Learning, there exists a methodology called *Reinforcement Learning* [Sut98, Wat89] that constitutes an efficient way to find solutions to problems in dynamic and unstructured environments, as well as executing complex tasks that depend on unforeseeable features of the environment.

In this work we inform about the state of the art in the the field of Self-

Reconfigurable Modular Robots, in order to determine the yet unresolved problems and possible niches of investigation of this field (chap. 2). Then we present the hypothesis we want to verify, and the tools that will be used for that aim (chap. 3). Amongst them there is a simulator entirely developed in order to study the use of Reinforcement Learning to obtain behaviours that efficiently solve locomotion and reconfiguration tasks on a M-TRAN Self-Reconfigurable Modular Robot. Finally (chaps. 4 and 5) we propose several experiments that involve the mentioned tasks and for which we study the best representation for the state-action space, the way to discretize this space in order for the experiments to be computationally feasible, and the impact of several aspects involved in a learning problem (definition of episode, learning velocity, evaluation of the learned policy). After designing and executing the experiments, we evaluate the obtained results and conclude about the effectiveness and possibilities of Reinforcement Learning when applied to this kind of robots (chap. 6).

Chapter 2

State of the art

2.1 Introduction

In this chapter we analyze the state of the art in the field of Self-Reconfigurable Modular Robots at the time we began this work.

We will see why this kind of robots are necessary, the benefits that can be obtained by using them, and the scenarios in which they can be used. Besides that, we will see why sometimes a more "classical" robot cannot be used, and what tasks are better suited for self-reconfigurable robots.

The rest of this chapter is organized as follows: in section 2.2 we state the main definitions and establish the capacities required by a Self-Reconfigurable Modular Robot. In section 2.3 we describe a possible taxonomy for this kind of robots. Section 2.4 deals with the most common problems encountered when working with this kind of robots. In section 2.5 we describe the most representative Self-Reconfigurable Modular Robots found in the bibliography. Section 2.6 explains how some of the presented problems have been solved by the different researchers. In section 2.7 we analyze in more detail some of the cited examples, and finally, in sections 2.8 and 2.9 we discuss some considerations and state our conclusions about this type of robots.

2.2 Main characteristics about Self-Reconfigurable Modular Robots

In this section we will discuss the main characteristics that describe Self-Reconfigurable Modular Robots.

2.2.1 What are Self-Reconfigurable Modular Robots?

- They are *modular*

  Modular robots are composed of many interconnected modules.

- They are *reconfigurable*

  A robot is said to be *reconfigurable* when it is possible to modify it's structure in order to adopt different configurations that allow it to perform different tasks.

- They are *self-reconfigurable*

  A robot is said to be *self-reconfigurable* when it can reconfigure itself by it's own means, without external intervention.

2.2.2 What makes Self-Reconfigurable Modular Robots so interesting?

Several aspects can be identified that make Self-Reconfigurable Modular Robots an interesting topic for research. In this section we will analyze the most relevant of them.

Robustness

> "Self-Reconfigurable robots are made from many identical modules and therefore if a module fails it can be replaced by another." [Sto04]

Since these robots are built from individual modules, it is possible to replace a defective module (or even separate it completely from the robot) so that it doesn't affect the behaviour of the robot. This characteristic allows the robot to degrade it's performance gradually instead of being rendered unusable by the malfunctioning of one module.

Versatility

> "The modules can be combined in different ways making the same robotic system able to perform a wide range of tasks." [Sto04]

A Self-Reconfigurable Modular Robot can acquire different configurations and use different kinds of modules in order to attain different objectives. In this way, it is possible to equip the robot with the best tools for each task it has to perform at the time it has to perform it.

While different tasks require different abilities, the robot remains the same (since modules are reused, even when the external structure of the robot varies, it will be composed of the same modules), but it can be prepared optimally for each new task at hand. This means that one Self-Reconfigurable Modular Robot can perform the same tasks for which several non-reconfigurable specialized robots would be required.

Naturally, this doesn't mean Self-Reconfigurable Modular Robots are the solution to every problem, because the production cost of these kind of robots might exceed the cost of several more specialized robots.

However, there are cases in which the utility of one Self-Reconfigurable Modular Robot greatly exceeds the cost difference. The most adequate scenarios for this kind of robots are those in which the environment in which they have to operate are dynamic and unstructured, and when there is very little (or none at all) information about the environment in which the robot will have to perform (for example in tasks like planetary exploration, or search and rescue in collapsed buildings).

In this kind of scenarios, the ability of these robots to acquire new configurations is critical. Due to the fact that it is not possible to accurately anticipate the environment, it is necessary for the robot to be able to adapt itself to the environment dynamically. Most specialized robots require that the environment is determined beforehand and be regulated according to strict specifications, and are therefore not adequate in these circumstances.

Cost

> "When the final design for the basic module has been obtained, it can be mass produced, thereby keeping the cost of an individual module low, compared to it's complexity." [Sto04]

Even though Self-Reconfigurable Modular Robots can be quite expensive to produce, due to the hardware complexity required to incorporate all necessary components in independent modules, and due to the complexity of the software required to solve the reconfiguration task, in many cases it can be more cost-efficient to use these kind of robots instead of several specialized robots in order to attain the same goals.

Besides that, the robustness of these robotic systems lessens the Total Cost of Ownership (TCO), since the same robot can be used in a diverse set of circumstances, avoiding the need of a specialized robot for each one of those cases.

Robustness, versatility and adaptability of these robotic systems provide them with a greater expected service lifetime than their non-reconfigurable

peers, since in traditional systems, in which several specialized robots interact in order to attain a goal, when one of these fails, it can force the entire group to abort the task. If this happens in a situation where it is not possible to recover the robots, the failure of the tiniest component would be as costly as the whole robotic system. It is in these cases when Self-Reconfigurable Modular Robots are clearly less costly.

Adaptability

> "While the self-reconfigurable robot performs it's task, it can change it's physical shape to adapt to changes in the environment." [Sto04]

Adaptability is a crucial factor when a robot has to operate in an environment that is dynamic and unstructured, in which operating conditions cannot be foreseen (or are stable enough).

In this situations, if the robot cannot adapt itself to the local conditions, it can become handicapped by external agents, without attaining it's objective.

Self-Reconfigurable Modular Robots perform particularly well in these kind of environments, since their ability to change their structure allows them to overcome unforeseen obstacles.

In an extreme case, it is even possible to detach part of the structure (that may have become unusable for some reason), and continue to operate (even though it might be in some restricted way).

Scalability

> "The size of the robot system can be increased or decreased by adding or removing modules." [Sto04]

Since Self-Reconfigurable Modular Robots are built from interconnected modules, it is possible to add more modules at some later point in order to allow the robot to attain configurations that require of a "greater size".

It is even possible for the robot to divide in smaller robots that can operate in parallel (and possibly perform different tasks), and later on reassemble in a single unit. This is specially useful for robots that perform exploration or patrolling tasks, since operating in parallel allows them to cover more surface.

Generally speaking, whenever a single robot has to perform tasks that can be divided in smaller sub-tasks, and where certain degree of parallelization exists, a Self-Reconfigurable Modular Robot can produce a significant difference in performance, by dividing itself in smaller robots that can each be assigned a sub-task.

2.2.3 What are the best scenarios for Self-Reconfigurable Modular Robots?

Search and Rescue

Any scenario in which the robot has to explore a certain location, like for example a collapsed building, in which operating conditions cannot be foreseen, can be an ideal case for these kind of robots. There could be rubble that the robot must go through; there could be holes and passages that lead to areas otherwise inaccessible, or there could even be places where the ground couldn't sustain too much weight. In this scenario, the robot might have to interact with people (in the case of finding survivors), objects (when retrieving objects from within the building) and the environment as a whole (in manipulating objects or moving obstacles).

A task like this would normally require several types of robots: a robot with good locomotion abilities, a manipulator robot, a robot that could interact with people, etc. The problem resides in that many of these specialized robots require controlled environments to operate correctly, a condition that most likely wouldn't hold in a scenario like the one previously described.

On the other hand, a Self-Reconfigurable Modular Robot could operate effectively in this kind of environment. If the ground were flat enough, it could first configure itself as a wheel, to move inside the building; at the presence of obstacles it could reconfigure itself to a more adequate configuration in order to overcome them. It might even find an opening that would allow it to pass to an area otherwise inaccessible, and it could reconfigure itself to pass through that opening, and once on the other side find out the best suited way to continue, according to the requirements of that new situation.

Once evaluated the situation, the robot might go back and let another team to continue the work; it might determine the necessary conditions for a more specialized robot, that would then have a predictable environment to operate in (having the environment being already assessed by the first robot). It might also be possible to send the same robot, but equipped with a different set of modules that would aid it to better solve the new situations encountered.

As mentioned, it can be seen that *Search and Rescue* is a perfect scenario for these kind of robots, where the environment cannot be determined beforehand (and can even change without notice).

Exploration

Exploration is another field apt for this kind of robot. From planetary exploration to undersea exploration, any unknown or unpredictable environment

constitutes a good opportunity to fully exploit these robot's capacities. In this kind of environments they can show the true potential of their abilities and successfully attain a goal that would be impossible to fulfil for other types of robots.

Again, it is the capacity to operate in unpredictable and highly dynamic environments and to adapt to changes in these environments what makes Self-Reconfigurable Modular Robots so exceptionally well prepared for exploration tasks. This capacity (to adapt to environmental changes) gives them a competitive advantage over their non-reconfigurable pairs.

It doesn't matter how well these robots perform in the mentioned scenarios, it is their ability to adapt and operate in a variety of circumstances that allows them to be used for nearly any task. They will of course not be as efficient as task-specific robots, but thanks to their versatility, this factor does not hinder their competitivity. On the contrary, they constitute a perfect choice in many cases, specially when there is a monetary limitation but it is required to use robots for several tasks, since it might be cheaper to acquire one Self-Reconfigurable Modular Robot than several specialized robots.

2.3 A classification for Self-Reconfigurable Modular Robots

The proposed classification is based on existing classifications, that distinguish robots according to several properties: *the classification according the homogeneity of the modules* [Rus00, Dit04, Cas02] and *the classification according to the spatial structure of the modules* [Dit04, Yos01b, Cas02]. We pretend to integrate both classifications into one that takes into account: homogeneity and spatial structure of the modules, as well as the reconfigurability of the robot. In this way it is possible to classify robots with a greater level of detail than by using only one of the previously existing classifications.

The proposed classification is structured according to the different properties taken into account:

- Homogeneity
  - Homogeneous
  - Heterogeneous
  - Hybrid (N-Homogeneous)
- Spatial structure

  - Chain-type
  - Lattice-type
  - Hybrid

- Reconfigurability
  - Reconfigurable
  - Self-Reconfigurable

It is worth noting that since this classification only takes into account Reconfigurable Modular Robots, those robots that are neither modular nor reconfigurable are not being considered.

2.3.1 Modular Robots

The main characteristics of Modular Robots are that they are composed of a set of independent interconnected modules, that can be either specialized, built specifically for a particular robot, or standard off-the-shelf modules, built for generic use.

Homogeneous Robots

In this type of robot, all modules are identical.

- **Advantages**

  Since all modules are identical, they can be mass-produced. This allows for a lesser total cost of the robot.

  Since there is only one type of module, the reconfiguration problem significantly simplifies, because the problem of selecting the correct type of module is no longer an issue (all modules can be selected, since they are identical).

- **Disadvantages**

  Each module must be self-contained, providing it's own energy, actuators and sensors. For this reason, this type of modules tend to be quite big, what generally mean heavier and makes them more expensive to produce and difficult to operate.

  Besides, each module must provide itself with all types of actuators and sensors that might be needed for the robot as a whole, which clearly isn't an optimal resource allocation strategy, because not all modules perform the same task (using the same sensors and actuators) simultaneously, and thus might share their resources.

Heterogeneous Robots

This type of robots can be built up from different types of modules, that each posses different characteristics.

- **Advantages**

  It is possible to specialize each module according to it's function, simplifying their manufacturing and reducing their cost.

  It is possible to add specific modules on demand, allowing the robot to assimilate them as if it would have been designed with those modules in the first place, giving it therefore greater functionality as it is required, and allowing for an incremental improvement process along the robot's service life. In this way, it is possible to use certain modules as a base configuration for the robot and later on add specialized modules according to the task at hand.

- **Disadvantages**

  The reconfiguration task is more complex, since having several types of modules involves having to take into account the number of each type of modules available, and how they interact with each other.

  Modular diversity restrains the number of modules of each type, and therefore one module can become a critical part of the robot, when no replacement for that module can be found in the remaining structure.

N-Homogeneous Robots

N-homogeneous robots represent an intermediate case between a homogeneous and a heterogeneous robot. This type of robots can be composed by modules of maximum N different types.

- **Advantages**

  The amount of different modules that must be taken into account is small (normally, between 2 and 3). This allows the reconfiguration problem to remain bounded and not turn out so complex as in the fully heterogeneous case.

  Since the amount of different modules is bounded, the construction of the robot is not as complex as in the fully heterogeneous case, and since there are several types of modules, each one can be specialized in a particular task, allowing them to be smaller, more efficient and cheaper to produce.

- **Disadvantages**

  It's not always possible to replace a module for some other one (since there are different kinds of modules).

  It's necessary to maintain a low heterogeneity for the complexity to remain bounded (otherwise this will constitute a case of a heterogeneous robot).

2.3.2 Reconfigurable Robots

A robot is said to be reconfigurable, when it is possible to modify it's structure in order to attain different configurations that allow it to perform different tasks.

- **Advantages**

  The reconfiguration ability of a robot makes it more versatile, since it can operate in a variety of situations.

- **Disadvantages**

  Robots that posses the reconfiguration capacity are more complex to build, and thus their cost is higher. Likewise, reconfigurable robots need to use a more complex software, which further increases the development cost.

2.3.3 Self-Reconfigurable Robots

A robot is said to be self-reconfigurable when it can reconfigure itself by it's own means, without external help.

- **Advantages**

  A self-reconfigurable robot possesses all advantages of a reconfigurable robot, and can operate in dynamic and unpredictable environments, thanks to it's autonomous nature given by it's independence of external help to reconfigure itself. This autonomy allows it to be used in environments where external intervention (mostly human) is not possible.

- **Disadvantages**

  A greater autonomy implies a greater complexity both in software as in hardware, resulting in a more complex and expensive manufacturing.

Chain-type Robots

In chain-type robots, modules are categorized into different roles, according to their location.

- *Branching* modules

  These modules are connected to at least three other modules.

- *Chain* modules

  These modules are connected to exactly two other modules.

- *End* modules

  These modules are connected to exactly one other module.

Reconfiguration is achieved by changing a module from one type into another (attaching or releasing other modules), until the desired configuration is reached.

Lattice-type Robots

In this kind of robots, modules interconnect in a lattice structure. Reconfiguration is achieved by changing the modules location within that structure, until the desired configuration is reached.

Hybrid Robots

The case of the M-TRAN robot (see section 3.2) is a special one; this a a robot that has both the qualities of a chain-type robot as well as those of a lattice-type robot. For that reason it's necessary to include the hybrid category in this classification.

Chain-type vs Lattice-type Robots

Lattice-type robots are better at solving the reconfiguration task than their chain-type counterparts. In the latter case, a module chain has to bend and dock with the rest of the robot. This process involves several modules and is hard to control.

The advantage of lattice-type systems is that just a few modules are involved in the reconfiguration process. However, even when these systems are better at the reconfiguration task, the current implementations are not very efficient. This is basically due to the difficulty of coming up with an efficient

connection mechanism. One of the most successful approaches is found in the implementation provided by the M-TRAN robot [Kur02, Mur02].

Other interesting systems include Telecube [Suh02], Molecule [Kot98] and I-Cubes [Üns99], but there haven't yet been enough modules built in order to evaluate these systems on a big scale [Sto04]. The other evaluated robots (see section 2.5) are bidimensional, which notably simplifies the reconfiguration task.

Chain-type self-reconfigurable robots have a greater degree of mobility than lattice-type robots. The reason for that is that generally speaking the DOF of chain-type robots are less restricted than in the case of lattice-type systems. If we analyze the differences between those systems it can be seen that chain-type systems are designed to achieve locomotion using a determined shape and eventually reconfigure to chain their shape. Lattice-type systems on the other hand are designed mainly for reconfiguration. An exception to this is the M-TRAN robot, which constitutes a middle-case between a chain-type and a lattice-type robot.

2.4 Common problems

In this section we discuss some of the more commonly found problems with Self-Reconfigurable Modular Robots. Since this type of robots is still in an early stage of research, most of the problems relate with basic tasks like locomotion and reconfiguration.

2.4.1 Reconfiguration

The first problem a Self-Reconfigurable Modular Robot has to solve is the one that gives it it's main characteristic: reconfiguration (although, strictly speaking, self-reconfiguration).

The first question that arises when trying to solve the reconfiguration problem is: how to achieve it?

In order to change it's structure, the robot must have a "notion" of the form (structure) it currently has, the configuration it wants to reach, and the steps needed in order to complete the transformation.

Thus, one of the main issues that has to be solved in order to attain reconfiguration is how to switch between two determined configurations.

Other problems specific to the reconfiguration task are:

- Docking
- Connection mechanism

Docking

This problem mainly affects chain-type robots. This type of robots need to dock with themselves in order to change a module from one type into another. Since docking must happen automatically and autonomously (without external intervention), this constitutes one of the main problems that need to be solved.

In order to dock, there must be a way for two modules to "sense" and "recognize" each other, so that they can connect to each other (it is necessary to keep monitoring the whole process to control the delicate movement required to align the modules so that the connection mechanism can be effective).

This requires some method to sense other modules, and some way to perform directed and precise movements. This is a non-trivial requirement, since it imposes restrictions both on hardware and software (which makes the whole robot's development more complex).

Connection mechanism

Another factor that has to be taken into account is the connection mechanism that is used between modules.

A good connection mechanism must be as simple as possible, and at the same time, robust enough to guarantee that the modules will not break apart unless intended. [Nil02]

Another important aspect to take into account when designing a connection mechanism is the maximum force the mechanism can tolerate. This can have a profound impact in the global performance of the robot, since a mechanism that doesn't allow the robot to lift it's own weight will certainly limit the reconfiguration possibilities.

There are several proposals for connection mechanisms, some using mechanical properties [Rus00, Nil02, Yim00, Cas02], some using magnetic properties [Yos01a, Pat04, Mur94, Suh02, Jor04, Kur02]. Many of the proposed mechanisms are not fault-tolerant, meaning that when one module fails there is no way to separate it from the rest of the structure. However there are some solutions that show to be promising in that aspect [Jan01].

2.4.2 Locomotion

Once the reconfiguration task has been solved, the next problem that arises is that of locomotion. This is due to the fact that locomotion is the first task a Self-Reconfigurable Modular Robot has to perform successfully in order to be used in real-world scenarios.

To achieve locomotion, the reconfiguration task must be solved, since in the case of chain-type robots, the robot has to acquire a configuration that will allow it to move around, while lattice-type robots must constantly reconfigure themselves in order to move.

So, this problem can then be divided into two cases, according to the robot's type.

Chain-type Robots

In chain-type robots, locomotion is normally achieved by following a predefined program for a given configuration, that will use a table of stored moves (*gait control table*) to guide to robots actions so that it will perform the desired movement.

The first issue with this approach is how to build that table, and how to store it, since depending on the robot's configuration, a simple "step" might involve actions on several modules. This, and the fact that a Self-Reconfigurable Modular Robot might acquire several configurations, results in the table's size to grow rapidly.

On the other hand, this also imposes a restriction on the hardware the robot must posses, in order to acquire locomotion abilities. These hardware requirements make the movements required to move a robot to become more complex, since more hardware is involved in each of those movements.

Because of this it is necessary to find a middle-ground between the robot's locomotion capacities and the hardware used to build it (and with that, the cost of the robot).

Lattice-type Robots

These type of robots move by constant reconfiguration. By changing each module's location, the whole robot can be transported to a new location, effectively performing locomotion.

In these kind of robots, the problem is that locomotion and reconfiguration really merge into one indivisible unit, which must be solved at the same time (a lattice-type robot that cannot move it's own modules cannot possible constitute a self-reconfigurable robot).

So, even if locomotion is a more "natural" ability to this kind of robots, they are generally more complex than their chain-type counterparts, and thus more complicated to build.

Most of the lattice-type robots are used as cellular-automata, in which each cell represent a module, and therefore they are (most of the time) programmed using cellular automata theory (by specifying rules that dictate the

movement of each module for each time period).

Naturally, in order to store those rules in the robot itself, it is necessary to have enough memory, and since these rule-sets are almost never small, and tend to become larger as the movements get more complex, the amount of memory needed becomes an inconvenient.

While used memory increases, the access times needed to determine to correct rule and execute it get longer, and therefore affect the performance and the real-time capabilities the robot might present. As can be seen, this problem also apply to chain-type robots (where the increase in the gait control table's size produces the same effect).

2.4.3 Methodology

Even if this does not really constitute a problem by itself, in this section we will discuss some of the different methodologies used by researchers when developing controllers for the locomotion and reconfiguration tasks.

Methodologies can be classified into:

- Planning
- Evolution
- Learning

Planning

The most widely used methodology tends to be planning (that is, to determine beforehand the robot's reaction to every possible situation). Most of researchers using this technique, used gait control tables to encode the necessary movements for locomotion [But02, Jor04, Kur02, Yim00]. This approach consists of assigning a set of actions (possibly in some specified order, for example sequential) to a state, so that whenever the robot finds itself in that state, it will perform the indicated actions. Normally, associative structures like tables are use to store these relationships, and hence the name.

Several works deal with how to apply this technique to various aspects involved in the reconfiguration task. Some of them study the problem of planning the immediate reconfiguration movement sequence [Yos01a, Yos01b, But02], while others use planning to determine the movement on a global level, like for example, trajectory planning [Yos01a, Yos01b].

Planning appears to be a successful strategy for dealing with the problems involved in this task, but somehow forces the developers to "prepare" the

robot's reaction in advance (by utilizing a previously designed movement table, for example), and impose hard restrictions on the robot's hardware (since most planning algorithms require of considerable computing power), so that they might even risk to go against the basic principles of Self-Reconfigurable Modular Robots (i.e, being autonomous and self-contained).

Evolution

Another approach was using evolution (as in genetic algorithms). Some authors decided to let the controllers evolve, in order to produce better controllers than could be obtained by alternative ways (like planning-generated or hardcoded controllers) [Kam03].

The evolutionary approach allowed the researchers to lessen the bias introduced into the controllers, and by letting the genetic algorithms to determine the optimal controllers, they could increase the robot's performance, thus reducing the cost-performance ratio.

This methodology however has the inconvenient that in order to obtain a sufficiently good controller, the genetic algorithm must run for a considerable amount of time (which diminishes the possibilities of using the robot in real-time scenarios). Beside that, since the *fitness function* must be modified each time the robot's physical structure changes, this will make it necessary to regenerate the controller. This also affects the robot's ability to adapt itself to the environment in a *fast* and *responsive* way.

Learning

Surprisingly enough, this methodology seems not to have been explored. The learning approach has the advantage of allowing the robot to adapt continuously to the environment and learn from the interaction.

From the robot's perspective, a change in the perception of the environment can be due to a real change in the environment, as well as to a change in the robot's perception capabilities. Thus, if the robot is able to cope with situations in an highly dynamic environment, it will also be able to modify it's behaviour to adjust for different perception capabilities (like when they decrease, for example, in the case of some sensors breaking).

This characteristic will allow the robot to be used in the real environment from an early stage in development, and evolving software and hardware simultaneously.

By starting to learn from early stages in the robot's life-cycle, another aspect gained is being able to verify the robustness of the robot's physical implementation, as well as of it's controller. It would be possible to produce

changes in the environment and evaluate how well the robot reacts, and at the same time, the robot would be learning to cope with that kind of changes. On the other hand, development and testing would be happening simultaneously, effectively reducing development time, and therefore, it's cost.

Naturally, no methodology is ever perfect, and learning has it's owns drawbacks. The mayor one is that it is necessary to invest a great amount of time in order to learn a behaviour that solves a given task efficiently (but since learning happens since an early stage in development, part of the invested time is actually gained). Another inconvenient is that in order to learn, more computing power is required than, for example, using gait control tables (that allow a robot the produce a predefined and limited behaviour), but at the same time will allow the robot to acquire more complex behaviours and even continuously refine them (without even mentioning the fact that it would be able to adapt to changes in the environment).

As could be seen, since a change in the environment cannot be told apart from a change the perception capabilities of the robot, by allowing it to continuously adapt to the environment, the learning methodology embodies the robot with the ability to cope with unexpected events and handle unforeseen situations or even react to events like the failure of some hardware components.

2.5 The contestants

In this section we will name the main characteristics of several existing Reconfigurable Modular Robots (and some self-reconfigurable ones).

2.5.1 Chain-type Robots

CONRO

This robot was developed in the year 2000 by Will and Shen at the Polymorphic Robotics Laboratory of the University of Southern California's Information Sciences Institute (USC/ISI). This is a homogeneous robot whose modules measure 10cm in length and weight 115g approximately. Each module, as can be observed in figure 2.1, is composed by two motors (standard radio-control equipment servomotors), two batteries, CPU (stamp II microcontroller) and IR sensors. It possesses a bipartite connection mechanism (1 female and 3 male connectors per module). Modules have two DOF that allow them to perform vertical (pitch) and horizontal (yaw) movements. IR sensors are used for data transfer as well as inter-module communication.

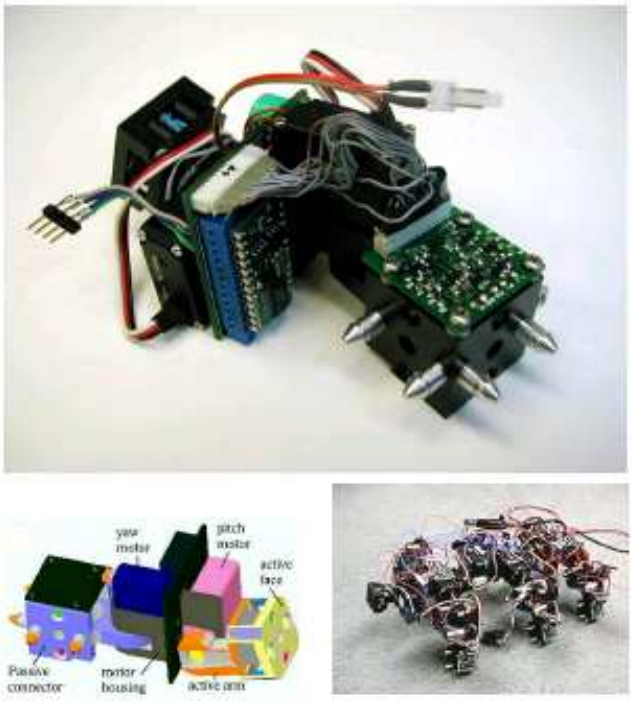

Figure 2.1: CONRO module (top). CAD model of a CONRO module (left). A hexapod robot built using CONRO modules (right). (USC Information Sciences Institute)

Polybot

This robot was developed at the Xerox Palo Alto Research Center in the year 2000, by Yim et al. Since then, three generations of this robot have been designed (G1, G2 and G3). This is a homogeneous robot, in which each module has one motor (in the G3 version, this a modified version of a 32mm Maxon motor), a battery, CPU (Motorola PowerPC 555, for the G3), and IR sensors (used for inter-module communication). The connection mechanism for these modules is based on hermaphrodite connectors. Figure 2.2 shows a picture of the G2 version.

ACM

The ACM (Active Cord Mechanism) robot is a homogeneous robot that was developed by Hirose et al. in 1993 at the Tokyo Institute of Technology. It was mainly used for simulating serpent's movements. Even if this robot can be used for manipulation and locomotion, and operates in 3D, this robot

Figure 2.2: CAD model of a Polybot G2 module (left). Polybot G2 robot configured as a wheel (right).

does not posses the ability of self-reconfiguration.

Figure 2.3: ACM-R1 robot.

2.5.2 Lattice-type Robots

Metamorphic Robotic System

This robot was developed during 1994 by Chirikjian, at the Johns Hopkins University, and then continued by Chirikjian and Pamecha in 1996. This a homogeneous robot composed of hexagonal modules (as can be seen in figure 2.4), each of which has 3 DOF (actually the modules consist of six interconnected rods, with actuators every two joints). Even though each module can connect to, disconnect from, and rotate around another modules, the reconfiguration possibilities are limited to 2D (since actuators operate at a bidimensional level). Each module has male and female connectors (and therefore this is a bipartite connection mechanism), but despite of this, there

is no way for two connectors of the same gender to dock (which greatly simplifies the docking task). Even though a self-reconfigurable robot must have it's own CPU, in the work published in 1996, the CPU wasn't yet integrated in the module, and an external controller was used (Motorola 68HC11).

Figure 2.4: Two Metamorphic robot modules.

Crystalline

The Crystalline robot, developed by Rus and Vona at the Dartmouth College in the year 2000, is another example of a homogeneous robot. Its modules are composed of a CPU (Atmel AT89C2051), an IR communication system and a battery and it uses a bipartite connection mechanism. Reconfiguration is achieved by expanding and contracting the different modules and by changing the connections between modules. Despite it's design being planar, it is possible to extend it to three dimensions. A Crystalline module can be seen in figure 2.5.

Fractum

Developed by Tomita and Murata in the year 1999 at the National Institute of Advanced Industrial Science And Technology, this homogeneous robot consists of modules that each one possesses a CPU (Zilog Z80), an optical communication system, batteries and three spherical wheels. It's connection mechanism is bipartite and consists of 6 connectors (3 for each gender), using electromagnetic connectors as well as passive magnets. This robot, which can be seen in figure 2.6 can only adopt planar configurations (2D).

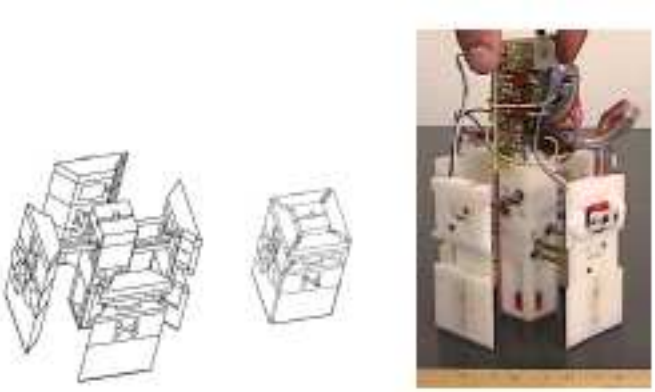

Figure 2.5: Model of an expanded and contracted Crystalline module (left). Crystalline module (right).

Figure 2.6: Three Fractum robots.

Micro Unit

The Micro Unit robot was developed in 2002 by Yoshida et al. at the National Institute of Advanced Industrial Science and Technology (AIST). This robot centers on miniaturization by using intelligent materials like SMA (Shape Memory Alloy), for connectors as well as for actuators. Its implementation can be observed in figure 2.7.

RIKEN Vertical

The RIKEN Vertical robot was developed in 1998 at The Institute for Physical and Chemical Research (RIKEN) by Hosokawa et al. This is an interest-

Figure 2.7: Micro Unit module (left). Structure formed by Micro Unit modules (right).

ing example, since even though this robot's reconfiguration capabilities are restricted to a plane, contrary to other similar robots, for this robot the vertical plane was chosen. Its modules have 2 DOF and use magnetic connectors. Figure 2.8 shows an example of this robot.

Telecube

This robot was developed by Suh, at PARC in 2002. According to what can be seen in figure 2.9, this is a three-dimensional version of the Crystalline robot. In this homogeneous robot, mobile magnets are used for docking.

MEL 3D Unit

The MEL 3D robot was developed by Murata et al. in the Mechanical Engineering Laboratory at AIST, in the year 2000. This homogeneous robot is capable of self-reconfiguration in 3D.

Molecule

This robot was developed by Kotay, Rus and McGray in 1998 (Dartmouth College). A module consists of two interconnected "atoms" with 2 DOF. This homogeneous robot is capable of three-dimensional reconfiguration. A picture of these modules is shown in figure 2.10.

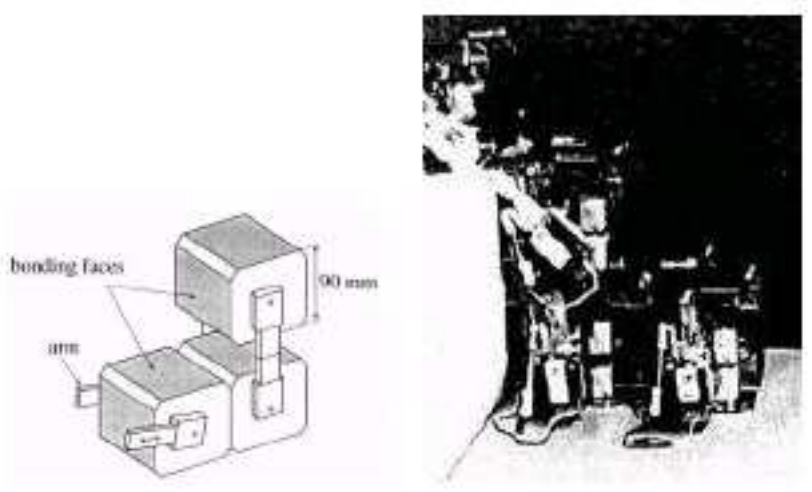

Figure 2.8: Model of RIKEN Vertical modules (left). RIKEN Vertical robot configured to climb stairs (right).

M-TRAN

The M-TRAN (Modular Transformer) robot, developed by Murata et al. during the year 2000 at the Intelligent Systems Research Institute of the National Institute of Advanced Industrial Science and Technology, is an example of a homogeneous robot whose modules have 3 connectors at each edge and posses 2 DOF. The connection mechanism is bipartite and allows it to perform three-dimensional self-reconfiguration. This robot will be further analyzed in chapter 3.

I(ces)-Cubes

I(ces)-Cubes was developed by Ünsal y Khosla in the year 2000, at the Institute for Complex Engineered Systems of the Carnegie Mellon University. This robot characterizes by being one of the few heterogeneous cases. There are two types of modules: the "link" module, which measures 80mm long and weights about 370g, possesses three DOF, as well as two "male" connectors, that can manipulate "cube" modules (which are entirely passive), which contain exclusively "female" connectors on each side. This robot is capable of self-reconfiguration. A pair of "cube" and "link" modules can be seen in figure 2.11.

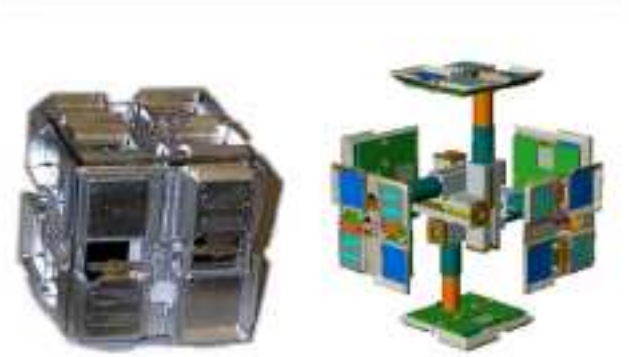

Figure 2.9: A Telecube robot (left). CAD model of a Telecube module in expanded state (right).

Fracta

This robot (shown in figure 2.12) is the three-dimensional version of the Fractum robot. It was developed around 1998 by Murata et al. at the Mechanical Engineering Laboratory (AIST). This homogeneous robot was the first system capable of self-reconfiguration. With it's cube-like modules measuring approximately 25cm long and weighting 7Kg, this 12-DOF per module robot, clearly shows the typical problems found in homogeneous systems: modules tend to be big and cumbersome.

Proteo

Proteo was developed by Bojinov et al. in the year 2000 at the Xerox Palo Alto Research Center. This homogeneous robot consists of rhombic-dodecahedron-shaped modules, with 12 identical connection faces. For the connection mechanism it uses electromagnetic connectors that allow it to perform movements by a combination of rotations of the modules around the edges of their connection faces. Having 12 connection faces involves a great complexity and high cost. There doesn't yet exist implementations of this robot, and the whole work about it has been done in simulations.

Miniaturized Self-Reconfigurable Robot

The Miniaturized Self-Reconfigurable Robot, developed at the Mechanical Engineering Laboratory (AIST), was presented by Yoshida et al. in 1999. This is a homogeneous robot of very little size (modules measure 40mm high, 50mm wide and weight 80g). The system was designed so it could use SMA

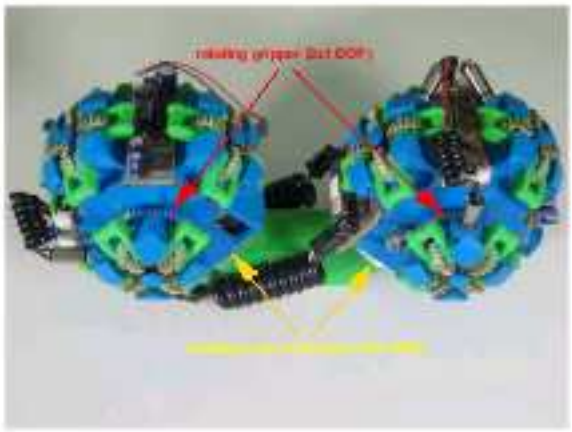

Figure 2.10: The Molecule robot.

actuators that allow it to reduce it's size. The connection mechanism is a bipartite one. It's reduced size only allows it a limited torque, thus reducing the range of movements this robot can perform.

Semi-Cylindrical Reconfigurable Robot

Another homogeneous robot capable of operating in 3D is the Semi-Cylindrical Reconfigurable Robot. This system, presented by Kurokawa in the year 2000 (developed at AIST), consists of modules formed by two semi-cylindrical cubes having each one servo-motor. The connection mechanism uses magnetic connectors as well as SMA springs in order to undock. The small magnets give it a limited strength and it's reconfiguration capacity is bounded by the bipartite nature of the connectors. Nevertheless, this robot is capable of self-reconfiguration in 3D.

TETROBOT

Developed by Hamlin and Sanderson in 1996 (Rensselaer Polytechnic Institute), TETROBOT is a homogeneous robot, that introduced a novel design by using spherical joints. This robot, which can be seen in figure 2.13 requires for the reconfiguration to be performed manually.

CEBOT

The Cellular Robotic System (CEBOT) was developed by Fukuda (Nagoya University) and Kawauchi (Science University of Tokyo) in 1990. This is a

Figure 2.11: "cube" and "link" modules for the I-Cubes robot.

homogeneous robot of cellular design that has limited sensing and computing capabilities. This robot is not capable of self-reconfiguration.

2.5.3 Comparison

As can be seen in table A.1 included in the appendix A, most of the research on reconfigurable robots has been done with homogeneous systems, with the aim to operate in three dimensions.

In most cases, modules can move around their neighbouring modules, and a robot built using these type of modules is capable of self-reconfiguration.

The amount of DOF varies between 0 and 12, being 2 DOF the dominant choice.

There doesn't appear to be a fault-tolerant connection mechanism yet. Most of the connection mechanisms depend on the module to be able to detach itself [Jan01].

The size of the modules varies greatly (from the biggest, as Fracta, to the tiniest like Micro Unit or Miniaturized Self-Reconfigurable Robot), but on average their size lies in the 40mm - 80mm range (measuring width).

2.6 The Solutions

In this section we discuss some of the proposed solutions to the problems described in section 2.4.

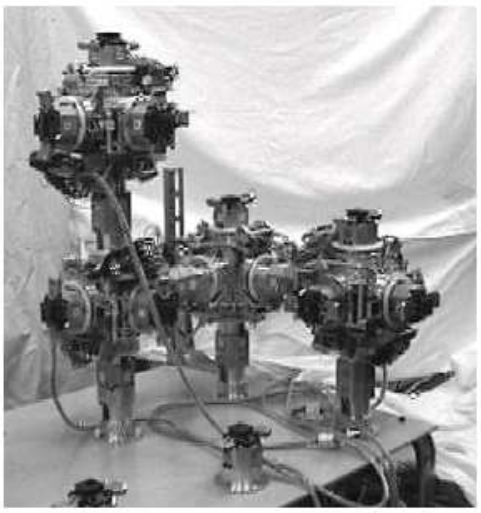

Figure 2.12: Fracta robot modules.

Figure 2.13: TETROBOT.

2.6.1 Reconfiguration

In most of Self-Reconfigurable Modular Robots researchers tried to solve the reconfiguration problem by using gait control tables.

According to this approach, the robot only has to determine which entry in those tables corresponds to the actual state, and execute the actions stated in that entry.

This is a simple and efficient solution, but nevertheless has a pretty important inconvenient: the gait control tables must be built on advance, and somehow transferred to the robot.

Since in this case reconfiguration turns out to be just a lookup, the computing time cost of this process is relatively low.

This way of handling the reconfiguration problem seems to be adequate for chain-type robots, where, as seen before, reconfiguration is independent of locomotion, and therefore shouldn't be required to be done frequently (the robot only needs to reconfigure when it is necessary to acquire a different locomotion gait, and therefore the amount of states involved in the reconfiguration task is low, thus bounding the size of the gait control table).

Another used approach is evolving that table [Kam03]. This approach is less popular, but not less interesting, since it tries to avoid some of the limitations of the previous approach. Through evolution, the robot is able to operate in a variety of scenarios that may not have been anticipated, since it is able to build it's own gait control table on demand, whenever a situation arises for which the robot doesn't have a predefined solution.

With this last method it will be necessary to wait for a certain amount of time before the robot can perform as good as by using pre-generated control tables, but since the robot can be used even before it has completed the table generation, the invested time in generating the complete table can be equivalent to the time it would take a person to develop a correct and complete gait control table.

Docking

An interesting example is the case of the Polybot robot, where researchers developed a mechanism that allows the robot to dock by it's own means, only using a couple of IR sensors [Rou00].

This example shows that even though the docking problem is not an easy task, since it involves controlled movement, sensing and "acknowledging" the "destination" module amongst several things, there are implementations that solved this task reasonably well.

Another solution for this problem is using magnetic connectors [Yos01a, Pat04, Mur94, Suh02, Jor04, Kur02] so that once the modules get near enough to each other, the magnetic forces will attach them automatically. This is a pretty clever solution, but requires that the modules be driven close enough for the magnetic forces to be strong enough to draw the modules together.

Connection mechanism

The connection mechanism is an indispensable aspect of a Self-Reconfigurable Modular Robot, and therefore constitutes one of the central issues in it's design.

Most proposed solutions are based on some form of mechanical coupling, using bipartite mechanical connectors (male/female) [Jor04, Sto02, Kur02, Cas02]. Even though this is a simple and straightforward approach, it has some inconvenients. The bipartite paradigm reduces the reconfiguration possibilities, because a module can only connect to modules of the opposite gender. If this is not possible (for example, when there are no modules of the opposite gender nearby, due to the current configuration), the robot will not be able to change it's configuration. On the other hand, most mechanical connectors can only be controlled by the module, thus when one module fails it cannot be detached from the rest of the robot (maybe thus degrading the robot's performance).

Another attempt to solve this problem consists in using magnetic connectors [Yos01a, Pat04, Mur94, Suh02, Jor04, Kur02]. In this case, the connectors are more robust (magnets do not fail) and the bipartite restriction can be avoided by using electromagnets (that can change their polarity). Electromagnets however, require the module to function correctly, since it is necessary to get electricity to flow through the magnet in order to generate a magnetic field. In the event of power shortage in the module, it is still possible to detach it, but nothing prevents the module to be unable to detach itself even when it has a power supply (for example, if the circuit responsible for attaching/detaching fails, it might be impossible to revert the electromagnets polarity).

Besides using passive magnets, some designers resolved to use SMA coils (*Shape Memory Alloy*), that can produce enough force to separate the magnets so that detaching of the modules is possible. A benefit of these kind of coils is that they are small and require very little power to operate.

2.6.2 Locomotion

Since locomotion is so closely bound to the reconfiguration problem, the proposed solutions for the latter issue also solve the problems that might appear during locomotion.

Besides, normally the mayor issue during locomotion is to have the hardware required (e.g. memory) to store the big control tables or rule-sets for the cellular automata used to guide the movements. Since these hardware issues are common to all kinds of robots, they are not further studied here.

2.7 The outstanding

In this section we will further analyze the robots we consider to be the most successful amongst the ones proposed in section 2.5.

The brief insight on each of these robots only serves the purpose of motivating the reader to consult further bibliography regarding each one of them. It is not our goal to provide a detailed analysis of the capabilities or limitations of each one of the mentioned robots.

2.7.1 M-TRAN [Mur02, Yos01a, Yos01b, Kam03, Kur02, But02]

This constitutes one of the most successful cases of a Self-Reconfigurable Modular Robot. One of the reasons for that is that this robot is a hybrid between chain- and lattice-type robots. This characteristic gives it the typical benefits of both types of robots and reduces the impact of their respective limitations.

An M-TRAN robot can operate (in regard of it's morphology) as a chain-type robot and perform locomotion without reconfiguration, or as a lattice-type robot that reconfigures itself continuously in order to move around.

Its hardware is quite simple, and thus can be relatively cheap to produce. Given it's size, it doesn't have many physical limitations, but the fact that it cannot generate too much force with it's motors (thus restricting it's capability for lifting heavy weights).

For it's connection mechanism it uses passive magnets for coupling, and SMA coils for decoupling. This provides it with a great flexibility for reconfiguration.

Even though this robot is not as popular as some other robots presented in this section, it has certain advantages of the other robots that positions it in the top rank. This advantages include that thanks to it's simple and relatively cheap hardware it can be used in situations in which due to the economic factor, the other options must be discarded. Another advantage is that in being hybrid it can operate as a chain-type or lattice-type robot, allowing experimentation on both research fields, without requiring two different robots.

Despite that, this robot has been mostly considered as a lattice-type robot, and therefore the majority of research using this robot has been focused on cellular automata theory for locomotion [But02].

Nevertheless, it can be easily seen that it's design allows it to perform locomotion with reconfiguration, for what both aspects can be explored simultaneously, allowing researchers to find an equilibrium between both designs.

2.7.2 Polybot [Yim00, Yim02, Rou00]

This robot can be considered to be the most popular one, and it has been found to be in constant improvement. Even though simulations and demonstrations with physical implementations have proven to show a very promising potential, this robot still shows one disfavourable aspect: it's hardware is not as simple as the M-TRAN's, and by being more complex, it's cost is higher.

The researchers at the Palo Alto Research Center (PARC) have been working on this robot for a long time now, and have continuously demonstrated this robot's potential.

Even though this robot may not fulfill the requirements of some research team, Polybot must not be ignored, as it's creators constantly show novel solutions to some real problems that might generate significant improvements within the field of Self-Reconfigurable Modular Robots.

2.7.3 CONRO [Cas02, Sto02, Sto03, She]

CONRO is a direct competitor for Polybot, but since it's hardware is the most complex one amongst the choices analyzed in this chapter we confer it the third rank.

It may be possible that it is not more expensive to manufacture than the other robots, but since their modules are bigger and heavier, it's usage opportunities are more restricted.

2.8 Discussion

Next we will introduce some ideas on how to improve the proposed robots, and will discuss some requisites a robot must fulfill in order to be more efficient.

2.8.1 Methodology

One of the main arguments of this section is that according the knowledge gained through the available bibliography, an aspect that has shown to be repeatedly ignored is the use of learning algorithms for locomotion and reconfiguration tasks.

This methodology has so many advantages that it is a surprise for us not to have found more references about it's usage in the field of Self-Reconfigurable Modular Robots.

Embedding the robot with learning capabilities would undoubtedly render the robot more robust and versatile (both highly wanted qualities in a Self-Reconfigurable Modular Robot).

2.8.2 Hardware design

Regarding hardware design, a successful robot must posses a hardware that is cheap to produce and easy to manipulate (to allow reconfiguration).

In order for a design to achieve these goals, several aspects must be taken into account. An optimal resource utilization and distribution must be guaranteed in order to maximize the sensor, actuator and available energy effectivity. On the other hand, the possibility for a module to fail must be contemplated, and it must be decided how to act upon such circumstances.

2.9 Conclusion

In this chapter we have briefly presented the state of the art in the field of Self-Reconfigurable Modular Robots, we have seen the main interest topics of it, and we have presented some of the most common problems and how they have been solved.

Several robots have been introduced, analyzing their main characteristics, when they showed some novel or interesting solution to a problem inherent to this field.

Then we proposed three robots as the most successful representatives of the field, briefly commenting their mayor virtues. We also reflected on the requirements a robot must fulfill in order to be successful, and we discussed some of the problems that the current designs had.

It clearly shows that this is still a young field withing the discipline of robotics. However, there already have been some important advances in it. There is still a long way to go before we can see Self-Reconfigurable Modular Robots operating successfully in the real world, performing the tasks for which they are best suited.

The appearance of Self-Reconfigurable Modular Robots marked a milestone in robotics. Their field of action is wide, and their use might be extended to a great variety of scenarios. It is just a matter of time before we can see this type of robots rescuing people from disaster areas, or exploring distant planets autonomously.

Chapter 3

Hypothesis, Materials and Methods

3.1 Objective

As mentioned in section 2.8, there was no mention in the bibliography of using learning as a methodology for solving locomotion and reconfiguration tasks in Self-Reconfigurable Modular Robots.

According to the knowledge of *Reinforcement Learning* and it's field of action, the hypothesis we want to verify is that Self-Reconfigurable Modular Robots can be given tools that will allow them to learn adequate end efficient behaviours for solving locomotion and reconfiguration tasks without human supervision.

We determined that the most appropriate robot for this work would be the M-TRAN robot, and that the experimentation would take place through simulations, for which it was necessary to develop a simulator that would contemplate all the characteristics of the problem: the simulation of an M-TRAN robot's body structure as well as the implementation of the learning algorithm.

In this chapter we will introduce some of the elements used along our work. Next, we will describe the properties and characteristics of the used robot, as well as the developed simulator and the implemented learning algorithm.

3.2 M-TRAN

M-TRAN (Modular Transformer) is a Self-Reconfigurable Modular Robot, developed jointly by AIST and Tokyo-Tech since 1998.

This robot's design presents the advantages of both reconfigurable robot types (chain-type and lattice-type), as seen in chapter 2. The hybrid design, and particularly the shape of the blocks that compose this robot's modules are key aspects that make this robot a very flexible system.

An M-TRAN module is composed by two blocks, interconnected by means of an axis (see figure 3.1). Each one of the three flat surfaces of these blocks is capable of connecting and attaching itself with the surface of another module's block. Since each block has a determined gender, the surfaces of a passive block can only attach to the surfaces of an active block, and vice-versa. However, having three connecting surfaces per block allows modules to connect to each other in several ways (see figure 3.2).

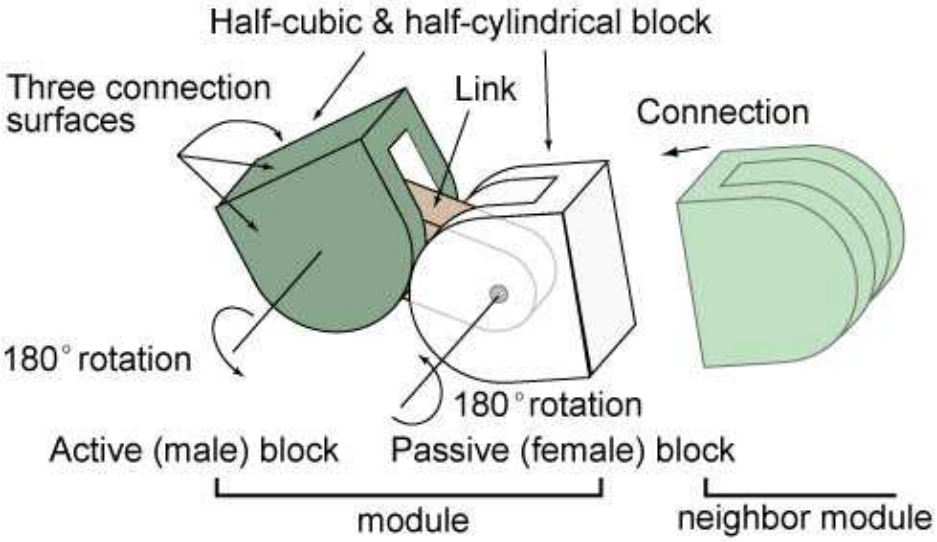

Figure 3.1: Schematic representation of an M-TRAN module.

Figure 3.2: Coupling between two modules, in several relative positions.

As mentioned, this design allows the robot to perform as if it was a lattice-type robot: if each block is orientated (with respect to the other blocks) in a way such that it's motors are set at angles of -90, 0 or 90 degrees, then all modules will be aligned in a three-dimensional grid-like structure.

On the other hand, this robot can also be used as a chain-type robot: if all motors (or at least several of them) are activated simultaneously, the robot can then perform movements with great flexibility.

Each of the M-TRAN's modules has it's own controller, which provides it with a certain amount of "intelligence" and autonomy. The different modules interact through their controllers, constituting as a whole a Distributed Autonomous System.

Currently, the M-TRAN robot sees it's third implementation. The first version was developed around 1998. That implementation was characterized by using magnetic connectors (see section 3.2.1), and was controlled remotely through a cable. In it's second implementation (2002), the most significant improvements were aimed towards a greater autonomy, by using batteries, a decentralized and distributed control and wireless communication to the modules (one-way link). This version was a little smaller than it's predecessor. Finally, the this version, built around 2005 got rid of the magnetic connectors, replacing them by mechanical connectors (see section 3.2.1), and provided the modules with full bidirectional wireless communication capabilities, as well as IR proximity sensors.

Figure 3.3: Different M-TRAN robot implementations: M-TRAN I (left), M-TRAN II (center) y M-TRAN III (right).

3.2.1 Connection mechanism

Along it's evolution, this robot used magnetic as well as mechanical connectors for the connection mechanism. Next we will detail both connector types, as they were implemented.

Magnetic connectors

This type of connectors was used in the first and second implementations of the M-TRAN robot. Each of the connection surfaces of the blocks of a

module has permanent magnets. In passive blocks, those magnets are placed directly on the connection surface, while in the case of active blocks, they are placed on a mobile structure named connecting plate.

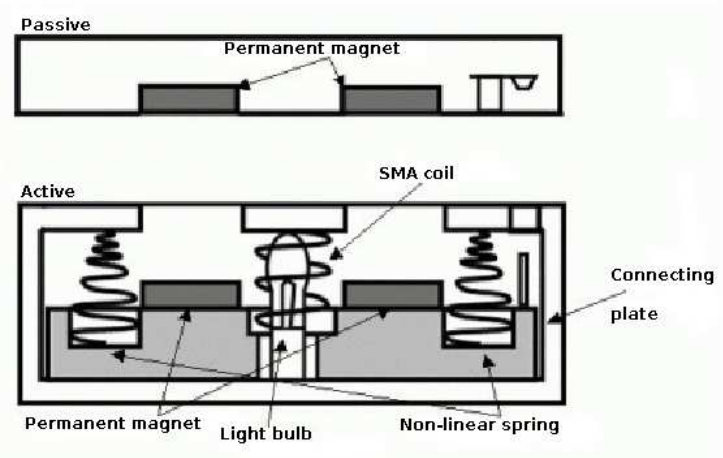

Figure 3.4: Magnetic connection mechanism.

The coupling/decoupling process goes like:

- Coupling
    1. The connection surfaces touch each other.
    2. The connecting plate is attracted towards the connecting surface due to the magnetic force. The non-linear springs don't provide enough resistance to counteract the magnetic force.
- Decoupling
    1. A small light bulb turns on. This lamp generates heat, which warms up the SMA coil around it.
    2. The coil expands due to the heat, and separate the blocks as far as needed for the magnetic force to be weaker than the repulsion force effected by the non-linear springs.
    3. The connecting plate separates from the connection surface due to the force effected by the non-linear springs.
    4. The light bulb turns off, and the SMA coil starts to cool down, allowing the coupling process to start over.

Mechanical connectors

Since the magnetic connection mechanism used in M-TRAN I and II robots was slow and consumed too much energy in order to control the connection process, it was decided to use a mechanical connection mechanism in the third version of this robot. That mechanism should allow the robot to perform the coupling and decoupling process in a more efficient and faster way than that of their predecessors.

Unfortunately, we couldn't find detailed information about the structure of this mechanical connector or of the mechanism controlling the coupling and decoupling process.

The only information we could gather about this connector are the figures shown next.

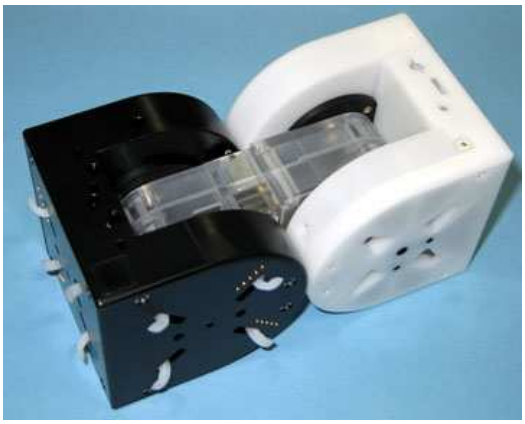

Figure 3.5: M-TRAN III module.

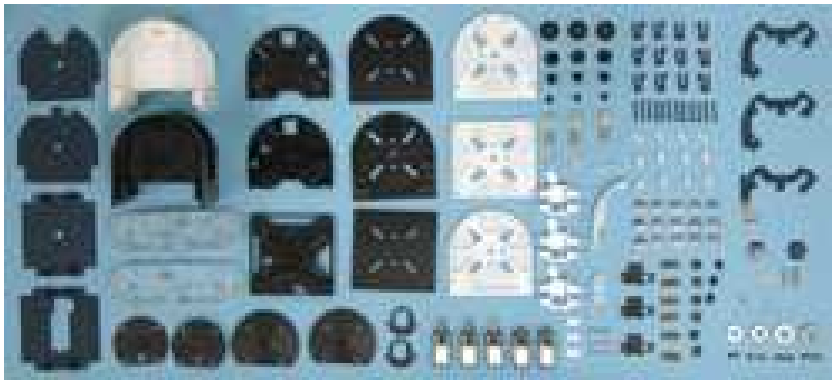

Figure 3.6: Mechanical parts that compose an M-TRAN III module.

According to what can be seen in these figures, the connection mechanism

uses latches in the active blocks that are inserted into holes in the passive blocks. The form of some of the pieces shown in the figure 3.6 suggests that once inserted, these latches are locked into position. However this is as much as can be speculated about this connection mechanism.

3.2.2 Pictures

Some pictures illustrating locomotion and reconfiguration tasks using the M-TRAN robot are included in the appendix C.

3.3 Simulator

For this work, a simulator capable of describing the structure and movements of an M-TRAN robot was developed. This simulator was implemented using a library for the simulation of rigid body physics, [ODE]. This means that even though the research was done on simulations, those simulations are relatively realistic, at least respecting gravity, the mass of bodies, elastic collisions, forces that interact between moving bodies, motor torques, etc.

The implemented simulator was designed with the intention of presenting the results in a as realistic as possible way. Therefore, the visualization cycle is an essential part of the simulation cycle, and the whole simulation takes place in "normal" time (by "normal" time we mean that the simulation time ratio is approximately 1 – or that 1 simulation second takes about 1 second to simulate).

The simulator was developed in a way that allows to work with robots that are composed by an arbitrary number of modules and configurations, as well as using different action-selection policies.

3.3.1 Implementation

Structure

The simulator was designed on a modular basis. This was so it can be easily changed and extended. The architecture is based on the following components:

- Simulation engine
- Robot
- Brain

- Configuration
- Scene description

The *Simulation engine* is the simulator core. This component implements the simulation algorithm and handles the event processing and visualization of the simulation.

The *Robot* component describes the structure of the simulated robot. A robot is composed of *modules*, which in turn are composed of two *blocks* and a *link*. The *link* component contains two *motors*, which control the block's movements. The structure is the same as the structure of the physical implementation of the M-TRAN modules.

The *brain* provides all the "intelligence"-related functionality. It is in this component where the action-selection policy and the learning algorithm are implemented.

The *configuration* and *scene description* components are both text files used for configuring each experiment. The *configuration* file defines the values of the units used to describe the robot, the environment, the interacting forces, etc. while the *scene description* file details the components that conform an experiment: the objects found in the simulated environment, like walls and the ground, and objects that form an active part in simulation, that is, the robots (it is possible to have more than one active robot). This file also describes the type of *brain* used (and with that, the used policy), as well as the modules that form each robot, and their initial configuration.

Learning policies

In order to implement the policies used during the learning phase of the performed experiments, an ϵ-greedy policy was used. This type of policy allows to combine stages of exploration with stages of knowledge exploitation. In order to reduce the amount of exploration as time increases, the ϵ value was modified according to a linear-decreasing function. This makes the exploration impact to be high on early stages of the experiment but lessening it as time passes by, effectively making the most impact to be due to actions decided upon the gained knowledge.

On the other hand, in order to be able to compare the effectiveness of the policies obtained by the learning algorithm, we also implemented a fully random policy and a policy that can reproduce a predefined action sequence.

To simplify the task of obtaining a policy by hand (in order to be able to compare it with the learned policies), we implemented a *brain* that is able to learn from a predefined action sequence. Using this brain, it is possible to actively teach the robot the desired behaviour. This constitutes a case

for supervised learning, that allows to learn a policy able to reproduce these predefined actions using just a few examples. This mechanism can be used, for example, to derive a policy for circular motion policy just form the actions needed to make a wheel-like configured robot make one complete turn.

Connection mechanism

In order to be able to simulate dockings, alignment and proximity conditions between modules were analyzed. If two modules are to dock with each other, and they are aligned and positioned in such a way that the coupling can take place, the simulator just takes the docking for successfully performed, without having to specifically simulate the full connection mechanism.

The conditions used to evaluate the docking viability resemble more the conditions required by the M-TRAN III robot, with it's mechanical connection mechanism than to the conditions required by the magnetic mechanism used by it's predecessors. This is due to the fact that the used physical simulation library doesn't provide the means to simulate magnetic properties (see 3.3.3).

In order to determine if coupling is feasible, the minimal distances between 4 points symmetrically located around the geometrical center of each connection surface are established. These points correspond to the positions of the holes and latches of the M-TRAN III connection mechanism. If the distance between each of the points in one of the connection surfaces and the points of the other module's connection surface do not cross a certain threshold, then the participating module's surfaces are considered to be correctly aligned and at a sufficiently small distance for the coupling to be successfully performed.

Once the modules have been coupled, the forces that interact to maintain the modules attached together allow for a slight misalignment to be possible. Nevertheless, any misalignment is automatically fixed by the simulation process.

3.3.2 Problems encountered and their solutions

During the experiments performed using the simulator, several problems were found. Next we will detail each one of them, along with the solution used to overcome them.

Tuning of the used units

To achieve a simulation as realistic as possible, the mass and size values for the modules and torque for the motors were defined as they were specified in several papers about the M-TRAN robot [Mur02, Kur02]. The M-TRAN robot research team was contacted by email in order to confirm these values.

The actual values used initially were:

Measure	Value	Unit	Notes
mass	400	g	full module
distance	60	mm	side of a module
torque	1.9	Nm	
gravity	9.81	m/s	
time	1	sec.	

Table 3.1: Initial values for the different units used during simulations.

These values couldn't be used directly as they were given, however, due to a limitation of the used physics simulation library. The limitation was that in order for the simulation to be as stable as possible, according to the library's documentation, it was suggested to adjust the units so that they would be within the $0.1 - 10.0$ range. Even after scaling all units (mass, distance, time, gravity, torque) consistently in order to fulfil this last requirement, simulations would not turn out satisfactory, because they would not "look" realistic. Therefore it was necessary to adjust some values by hand so that the simulations would look more realistic.

Resolution method for the equation system associated to the physical system

The physical simulation library used provides two methods for solving the equation system associated to the physical system. One of these methods, named WorldStep, uses a "big matrix" approach, trying to solve the the associated matrix in a traditional way (by calculating the inverse matrix). The other method, named QuickStep, uses an iterative algorithm to solve that matrix. The former method (WorldStep) uses a variation of the Dantzig method, also known as SIMPLEX, while the latter (QuickStep) uses an iterative algorithm that progressively relaxes the equation system's restrictions until it can be solved.

While the first approach produces more exact results (and thus nearer to the reality), it is highly sensitive to situations in which the associated matrix is near singular. In those cases, the simulation turns unstable (like in

numerical instability, meaning that the physical system tends to diverge as time passes, instead of converging to a stable state), producing unpredictable behaviours (that can be seen in the simulation as "explosions", or very strong collisions between the different simulated components, making each one to be propelled into a different direction at a great speed). The second approach is faster than the first and is not so sensible to singular matrices, but does not produce as precise results.

In this case we decided to use the WorldStep method, because the sensitivity towards singular-like matrices can be reduced by modifying certain parameters (actually, this is done by relaxing the definition of "singularity" as understood by the library). The library allows to modify a parameter named CFM (Constraint Force Mixing), that permits relaxing the restrictions imposed upon the physical system, therefore simulating amongst other things spring-like joints or spongy materials. Increasing the value of the CFM parameter also produces the effect of driving the system away from a singular state.

3.3.3 Limitations

As for the simulator limitations, they can be divided into two categories: the limitations due to the design and implementation of the simulator, and the limitations due the used libraries.

Limitations due to the simulator's design

The simulator was designed so that it should be simple to perform different experiments, but without going all the way into making it a general-purpose simulator. For this reason, in order to modify certain aspects of the simulation it turns out necessary to modify the source code itself. In spite of this, the maximum attention was drawn to implement it as generic as possible, and thus avoid having to modify the source code as much as possible.

The scene description (the environment used for the experiment) and the robots configuration (including the initial configuration for all blocks) must be specified manually in a configuration file. While this simplifies the execution of different experiments (one scene description file per experiment), defining the initial position for each block is a delicate and cumbersome process, since one tiny misplacement can prevent some blocks from being correctly attached, and therefore from achieving the robot's desired initial configuration.

Since the simulator was designed so that simulations would happen in "normal" time, the time taken for learning can be quite considerable (it is

not possible to simulate at accelerated time rates).

Limitations due to the used libraries

The simulator is not able to simulate the connection mechanism implemented by the M-TRAN robot (in it's I and II versions), because the physics simulation library used (ODE) does not provide magnetic properties simulation capabilities. However, in order to simulate the docking between modules, we implemented a "virtual" connection mechanism that just attaches two modules when the necessary conditions hold for such a connection to be feasible in reality, without having to simulate the specific connection process.

On the other hand, that library also doesn't provide the necessary primitives required to fully describe the physical body structure of the blocks. Because of this, we were forced to simulate those bodies as cubes, losing accuracy. This makes the simulation not to be completely realistic. Despite this, the used approximation is good enough to obtain indicative results for the robots behaviour.

Having to implement blocks as cubes forced us to disable the collision detection between the same module's blocks, in order to be able to rotate those blocks. Figure 3.7 shows how the cubes that represent the blocks collide while being rotated. Not having disabled collision detection between those bodies, it wouldn't have been possible to rotate them.

Having disabled the collision detection between the blocks, it was necessary to come up with another way to limit the block's rotation range (if it would have been possible to simulate the exact physical structure of the blocks, the collision detection mechanism would have automatically restricted the rotation range to 180 degrees). In order to restrain the rotation range, we had to restrict the possible angles the motors could take while they were set. That is, while figuring out the new desired angle for a motor, the same was limited so that it would remain in the $[-90, 90]$ range.

3.4 Learning

In order to learn an adequate locomotion or reconfiguration policy, we used Reinforcement Learning implemented by the Q-learning algorithm.

3.4.1 Reinforcement Learning

Reinforcement Learning covers a problem class within the spectrum of Machine Learning, in which an agent must explore an environment where it has to perceive it's own state, and can perform actions in order to modify it.

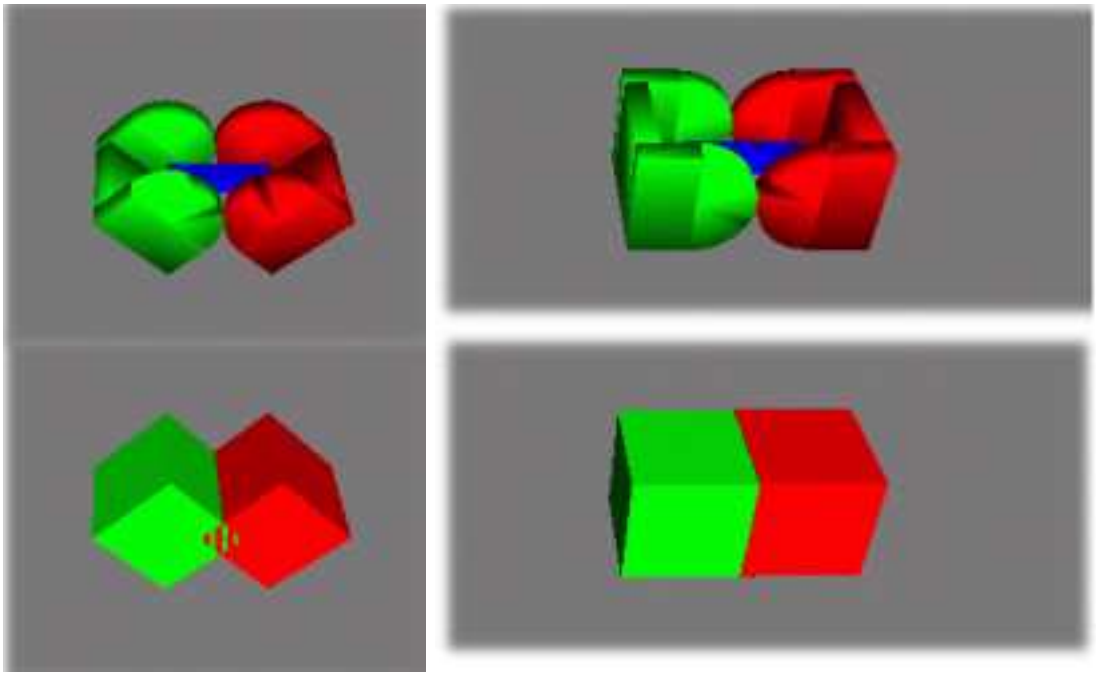

Figure 3.7: Real module structure (top) and simulated structure (bottom) visualization.

Normally, it is possible to formulate the environment through a Finite State Markov Decision Process (MDP), and the reinforcement learning algorithms used in this context are strongly related to dynamic programming techniques.

Reinforcement Learning Elements

Besides an agent and an environment, it is possible to identify three main elements in a Reinforcement Learning system: a *policy*, a *reward function* and a *value function*.

The *policy* defines the agent's behaviour at each moment (it is basically a relation between the states perceived by the agent and the possible actions to be performed in each state). Policy representations range from simple structures like lookup tables to complex processes the require a vast amount of computation.

The *reward function* defines the learning problem's goal. This is a function that assigns a numerical value to each possible tuple sas', indicating the inherent convenience to perform an action a while being on a state s and evolving to a state s'. The goal of a Reinforcement Learning agent is to max-

imize the long-term sum of the obtained rewards. The reward function must not be altered by the agent, since it represents the intrinsic characteristics of the problem at hand. Nevertheless it can be used to modify the used policy: whenever an action produces a low (or negative) reward, the policy can be modified so that that action is not chosen again in the future.

While the reward function tells which actions and states are "good" in short-term, the *value function* specifies the long-term optimal states. The *value* of a state is the total reward an agent can expect to accumulate starting from that state.

Reinforcement Learning Problem Formalization

Within the paradigm of Reinforcement Learning, the agent interacts with the environment at discrete time intervals $t = 0, 1, 2, 3, \ldots$. At each time instant t, the agent receives an environmental state representation $s_t \in S$, where S is the set of possible states, and selects an action $a \in A(s_t)$, where $A(s_t)$ is the set of possible actions for state s_t. At the next time interval, as part of the consequences of the performed action, the agent receives a reward $r_{t+1} \in \mathbb{R}$, and perceives a new state s_{t+1}. This interaction is shown in figure 3.8.

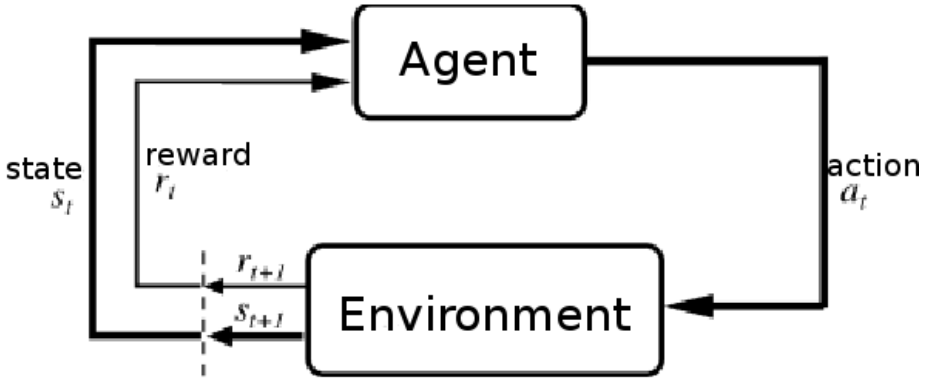

Figure 3.8: Agent-Environment interaction diagram for a Reinforcement Learning problem.

To determine the actions to perform, the agent uses a *policy* that relates each state to the best possible actions to be performed at that state. The Reinforcement Learning algorithms allow to obtain the optimal policy that maximizes the total accumulated reward.

Reinforcement Learning Algorithms

The Reinforcement Learning technique differs from supervised learning techniques by the fact that in the former the "correct" solutions are never specified, nor the "erroneous" actions explicitly corrected. The main focus of this technique is on the "on-line" performance, that involves finding a balance between exploration (trying out non-executed or less executed actions), and exploitation (of the acquired knowledge). The exploration-exploitation dichotomy has been studied thoroughly before, through problems like the *n-armed bandit* [Sut98] for example.

Thus, Reinforcement Learning is particularly well adapted to problems in which it is necessary to find a balance between immediate and long-term rewards. This technique has been successfully applied for the resolution of several problems, including robot control, elevator programming [Cri98], telecommunications [Kum98], backgammon [Tes95] and chess [Bax97].

Once the reward function has been defined, it is necessary to determine the algorithm used for finding the policy that maximizes the reward. There are mainly two approaches for that: the value-function method, and the direct approach.

The direct approach involves two steps:

a) For each possible policy, gather all rewards obtained by using that policy

b) Choose the policy that produces the maximum expected reward

This approach has basically two problems: on one hand, the amount of possible policies can be extremely large, or even infinite. On the other hand, the rewards might be stochastic, in which case it might be necessary to gather a very large amount of samples in order to precisely estimate the expected value for each policy. In spite of that, this approach has been used as a basis for the algorithms commonly found in evolutionary robotics [Nol98].

If we assume a certain structure in the problem that we are trying to solve, some of these inconvenients can be bypassed. The value-function approach is based on the concept that the rewards obtained by using one policy can affect the estimates made for another policy. In this approach the expected value when using a policy π is estimated, starting from a state s

$$V^{\pi}(s) = E_{\pi}\left\{R_t | s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right\}$$

or, estimating the expected value when taking an action a in a state s and then continuing according to a policy π

$$Q^{\pi}(s,a) = E_{\pi}\left\{R_t | s_t = s, a_t = a\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a\right\}.$$

Given a function Q for the optimal policy, it is possible to select the optimal actions just by choosing those actions that present a maximal value for each state s. In order to produce the same result with the function V, it is necessary to have a model of the environment, in the form of the probabilities $P(s'|s,a)$, that allows to calculate Q according to the equation

$$Q(s,a) = \sum_{s'} V(s') P(s'|s,a) \tag{3.1}$$

or, one of the so called "Actor-Critic" methods [Sut98] can be used, in which the model is split into two parts: the critic, that maintains information about the estimated value of the V function, and the actor, that is responsible for choosing the appropriate actions for each state.

Expanding the equation that defines function V^{π}, we obtain:

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\left\{R_t | s_t = s\right\} \\ &= E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right\} \\ &= E_{\pi}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \middle| s_t = s\right\} \\ &= E_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s\right\} \end{aligned}$$

Given a policy π, estimating $E_{\pi}\left\{R_t | s_t = s\right\}$ for $\gamma = 0$ is trivial, since we just have to take the average of the immediate rewards. The easiest way to do this for $\gamma > 0$ is to average the total reward, after each state is reached. This Monte Carlo style sampling requires however, that the associated MDP has terminal states.

Therefore, performing this estimation for $\gamma > 0$ in the general case is not obvious. However, it is actually simple, if it is noted that the expected value of R is a Bellman recursive function:

$$E_{\pi}\left\{R_t | s_t\right\} = r_t + \gamma E_{\pi}\left\{R_{t+1} | s_{t+1}\right\} \tag{3.2}$$

Replacing those expected values for $V^{\pi}(s_t)$ y $V^{\pi}(s_{t+1})$, it is possible to derive the temporal difference learning algorithm TD(0), whose update rule is described by

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

In it's simplest case, the set of states and actions are both discrete, and it's possible to maintain an estimate for each state-action pair. Other similar methods are SARSA and Q-Learning.

The aforementioned methods all converge to the correct estimate for a given policy, but they can also be used to determine the optimal policy. Normally, this is done by following a policy π that is derived from the estimated values at each time step (for example, by choosing most of the time the actions associated with a maximal Q value, but occasionally taking random actions, in order to explore other possible actions).

Q-Learning

Q-Learning [Wat89], as we saw in the previous section, is a Reinforcement Learning algorithm that uses the information associated to a state s' (i.e. $Q(s', a')$) to enhance the estimate $Q(s, a)$, by executing an action a when on a state s, following a give policy π. An important benefit of this algorithm resides in it's capacity to compare the expected reward for each of the possible actions, without having to have a model of the environment (see equation 3.1).

The central mechanism of this algorithm is based on the simplicity of the estimate updating rule. For each state $s \in S$, and for each action $a \in A$, the expected reward value is calculated according to the equation

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{3.3}$$

where, r_{t+1} is the reward obtained after performing the action a_t, when on a state s_t; α is the convergence rate (also known as learning rate), such that $0 < \alpha < 1$ and γ is the discount rate, so that $0 < \gamma < 1$.

The so learned state-action function Q directly approximates Q^*, the optimal state-action function, independently of the policy used. This fact radically simplifies the algorithm's analysis. The policy is still used to determine the actions to perform at each state, and only one requisite is necessary to guarantee the correct convergence: every state-action pair must be continually updated. This means that while a minimum of exploration is performed by the used policy, the algorithm guarantees that the Q function will converge towards Q^*.

This algorithm can be expressed as:

1. Initialize $Q(s, a)$ in some arbitrary way
2. **Repeat** for each episode
   (a) Initialize s
   (b) **Repeat** for each step (of the episode)
      i. Choose a for s, using a Q derived policy (for example ϵ-greedy)
      ii. Perform the action a, and obtain the reward r and the new state s'
      iii. $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)]$
      iv. $s \leftarrow s'$
   (c) **until** s is a terminal state

For the implementation of this algorithm some classes were defined: *Policy*, *ActionValueFunction* and *RLBrain*. *RLBrain* encapsulates the whole learning behaviour. In this class we define the best possible action a given a state s, through the used policy (implemented in the *Policy* class). To determine this, the policy has access to the information provided by the Q function, implemented by the *ActionValueFunction* class. This mean that *Policy* implements a policy derived from Q. Once the action a has been obtained and executed, in *RLBrain* we obtain the reward associated to that action (and state), and call a method that updates the Q table.

The halting condition, determined as "s is a terminal state" in the previous schema is defined in *RLBrain* (or any of it's derived classes), allowing that condition to depend on the performed experiment (this is also valid for the reward function, which in turn depends on the *RLBrain* derived class that was used for the performed experiment).

Chapter 4

Experimentation

In this chapter we analyze the impact of the Reinforcement Learning technique when applied to solving locomotion and reconfiguration tasks in M-TRAN robots.

The experiments will be performed in episodes. Unless noted, an episode is considered to be the execution of a simulation, according to the following definition:

Simulation The execution of the simulator program until a termination condition is met. Termination conditions depend upon the type of *Brain* (see 3.3.1) used. Some possible termination conditions are: the amount of executed actions; the evaluation of a condition, like for example, "distance to target is less than a critical value".

4.1 Locomotion

First, an experiment is performed with the aim of:

1. validating the developed simulator, by studying a real-world case
2. consolidating the methodology and knowledge needed to perform a learning experience with Self-Reconfigurable Modular Robots using the developed simulator

4.1.1 Experiment 1: Locomotion in a minimal-configuration robot

We try to solve a simple locomotion task, in a simple environment. The simulated environment has four walls, and naturally, a floor. The latter is

uniformly flat, and the environment is free of obstacles (not counting the walls). The robot used for this experiment consists of just one module.

For the learning task we use the *Q-Learning* algorithm, as it was presented in section 3.4.1.

Problem analysis

Since the experiment takes place in a simulated environment, but trying to maintain the maximum possible correlation to reality, it is necessary to avoid certain "benefits" of this kind of environments; one of them is the possibility of knowing the robot's position in absolute coordinates. Since a M-TRAN robot does not posses this ability, we have to find another way to determine the robots advance.

As was seen in section 3.4.1, the used algorithm involves the concepts:

- State
- Action
- Reward

In the case of *State* and *Action*, it is necessary to take into account that we try to find a definition of those concepts that keeps the amount of "state-action" pairs bounded. This is because the learning capacity and velocity depend directly on the size of the sets determined by these variables.

State

It is necessary to draw the difference between the notion of state as in the robot's perception capacities and as in the set of values needed for the learning process. For example, even if the robot is able to determine the current date, this data will be useless for learning locomotion. So, the date, even though it is part of the sensory state of the robot (because it can perceive this stimulus), does not belong to the set of variables that describe the state used for learning, since it doesn't add any information that helps to achieve the robot's goal.

Generally speaking, we can say that the state for the learning process is a subset of the sensory state of the robot.

Action

According to the stated goal, actions that the learning algorithm has to consider are strictly related to the robot's movement in the experimentation

environment. Since the robot has only one module, the available actions are restricted to those that a module can perform; these actions are limited to the motor's movements of the blocks that conform a module.

We try to simulate servo-like motors, that are controlled by specifying the desired target angle. Therefore, actions consists just of the angles that we want those motors to be in at each time step. In order for the learning task to be computationally feasible it is best to have as little angle choices as possible, while allowing all necessary angles for the robot to be able to move around the environment.

Reward

According to the stated goal, the reward function must favour those cases in which the robot moves in the desired direction, and penalize those case in which the robot doesn't reduce it's distance to the goal. It is important to try to avoid situations in which the robot might become immobilized.

Proposed solution

As was seen previously, it's necessary to avoid using absolute positioning. Therefore it's possible to put a light or radio source in the environment, and give the robot sensors that allow it to establish if it's getting closer or not to the emitting source, without having to know it's absolute position.

Given the mobility limitations of this kind of robot (being constituted by only one module limits it to a unidirectional movement), it must be initially aligned to the emitting source, in order to maximize the possibilities of the learning process.

Under such circumstances, the robots objective consists in moving in such a way as to minimize the distance between itself and the emitting source.

In order to implement this solution, it's necessary to define:

- Where to put the sensors?
- How many sensors to use per module?
- How to encode the sensor's values?

After analyzing some possibilities, it was decided to implement the solution described next.

Sensors (stimulus)

To determine the robot's advance, sensors are located in the mass center of each block. For simplicity's sake, sensors are omnidirectional (they measure the straight-line distance to the emitting source).

Sensors have a defined maximum sensing range, and divide the measurement spectrum into a finite amount of identical and equally-sized intervals. The sensor's value is determined according to the interval detected. The function used to calculate this value is given by equation 4.1.

$$v = discretize(|p_s - p_f|) \tag{4.1}$$

where

$$\begin{aligned}
discretize(x) &= \begin{cases} 0 & \text{if } x \geq range_s \\ \lfloor (1 - \frac{x}{range_s}) \times levels_s \rfloor & \text{otherwise} \end{cases} , \\
p_s &= \text{sensor's position} , \\
p_f &= \text{emitting source's position} , \\
range_s &= \text{maximum sensing distance} , \\
levels_s &= \text{number of the sensor's discretization levels} .
\end{aligned}$$

Even though the robot's and emitting source's positions are used in order to calculate the sensor's value, this data is encapsulated by the sensor and therefore constitutes an "implementation detail" of the sensor. The robot does never realize this information, but only uses the value provided by the sensor.

Robot (state)

The robot's state (according to it's perception capacity) is formed by 4 components. The robot is able to determine:

- the angle of each of the blocks (this information is provided by the block's motor encoders)
- the value perceived by each of the sensors located in the blocks

At the same time, the robot also possesses some "virtual sensors", by which it can determine:

- "pseudorotation" of each module

- direction to the emitting source

By "pseudorotation" we understand the value obtained from the relative position between the blocks that form a module, and that gives an idea of the module's rotation in space.

The way of calculating this value is given by equation 4.2.

$$v = pseudorotation(v_{active}, v_{passive}) \tag{4.2}$$

where

$$\begin{aligned} v_{active} &= \text{active block's sensor value ,} \\ v_{passive} &= \text{passive block's sensor value ,} \\ pseudorotation(x,y) &= \begin{cases} -1 & \text{if } (x-y) > 0 \\ 1 & \text{if } (x-y) < 0 \\ 0 & \text{otherwise} \end{cases} . \end{aligned}$$

On the other hand, each module can determine if the emitting source is closer to it's active or passive block, which determines the direction towards it must move.

In order to calculate this value, equation 4.3 is used.

$$v = normalize((d_{active} + d_{passive}) \times axis_s) \tag{4.3}$$

where

$$\begin{aligned} d_{active} &= \text{active block's direction ,} \\ d_{passive} &= \text{passive block's direction ,} \\ axis_s &= \text{initial block alignment ,} \\ normalize(x) &= \begin{cases} 1 & \text{if } x > 1 \\ -1 & \text{if } x < -1 \\ 0 & \text{otherwise} \end{cases} . \end{aligned}$$

These values can be obtained according to the following equations:

$$\begin{aligned} d_{block} &= map(direction, p_{block} - p_f) \ , \\ map(f,v) &= concatenate(f(head(v)), map(f, tail(v))) \ , \\ direction(x) &= \begin{cases} -1 & \text{if } x > 0 \\ 1 & \text{if } x < 0 \\ 0 & \text{otherwise} \end{cases} , \end{aligned}$$

$$
\begin{aligned}
p_{block} &= \text{block's position ,} \\
p_f &= \text{emitting source's position ,} \\
axis_s &= normalize(p_{i_{passive}} - p_{i_{active}}) \text{ ,} \\
p_{i_{block}} &= \text{initial block's position .}
\end{aligned}
$$

where *concatenate*, *head* and *tail* are the list handling primitive functions.

Again, in this case, even though the sensor value calculation involves the robot's and emitting source's positions, the robot does not have access to this information, but only to the value indicated by the sensor. If there was an alternative way of calculating these values, it would be possible to replace the sensor's implementation without affecting the robot's behaviour.

Learning (state)

As was previously seen, the robot's state and the state used for learning (the state used by the policy in order to determine the action to execute) are not necessarily the same. For this particular experiment it is necessary to draw the distinction and define the state used for learning as:

- $e_{robot} = e_{module}$
- $e_{module} = [\psi, \delta, \alpha, \beta]$
- $\psi =$ "pseudorotation"

  Possible values: $\begin{cases} -1 & \text{active block nearer to the emitting source} \\ 0 & \text{both blocks equally distant to the emitting source} \\ 1 & \text{passive block nearer to the emitting source} \end{cases}$
- $\delta =$ direction to emitting source

  Possible values: $\begin{cases} -1 & \text{to the "right" (positive coordinates)} \\ 0 & \text{parallel to the emitting source} \\ 1 & \text{to the "left" (negative coordinates)} \end{cases}$
- $\alpha =$ active block angle

  Possible values: -90, 0, +90
- $\beta =$ passive block angle

  Possible values: -90, 0, +90

This state definition allows to distinguish each possible configuration, through the angle values of each block. The ψ value gives information about the module's spatial rotation, which allows to determine which direction is meant by "forward", while the δ value gives information if the robot should move "forward" or "backwards".

Figure 4.1 shows several examples of possible states.

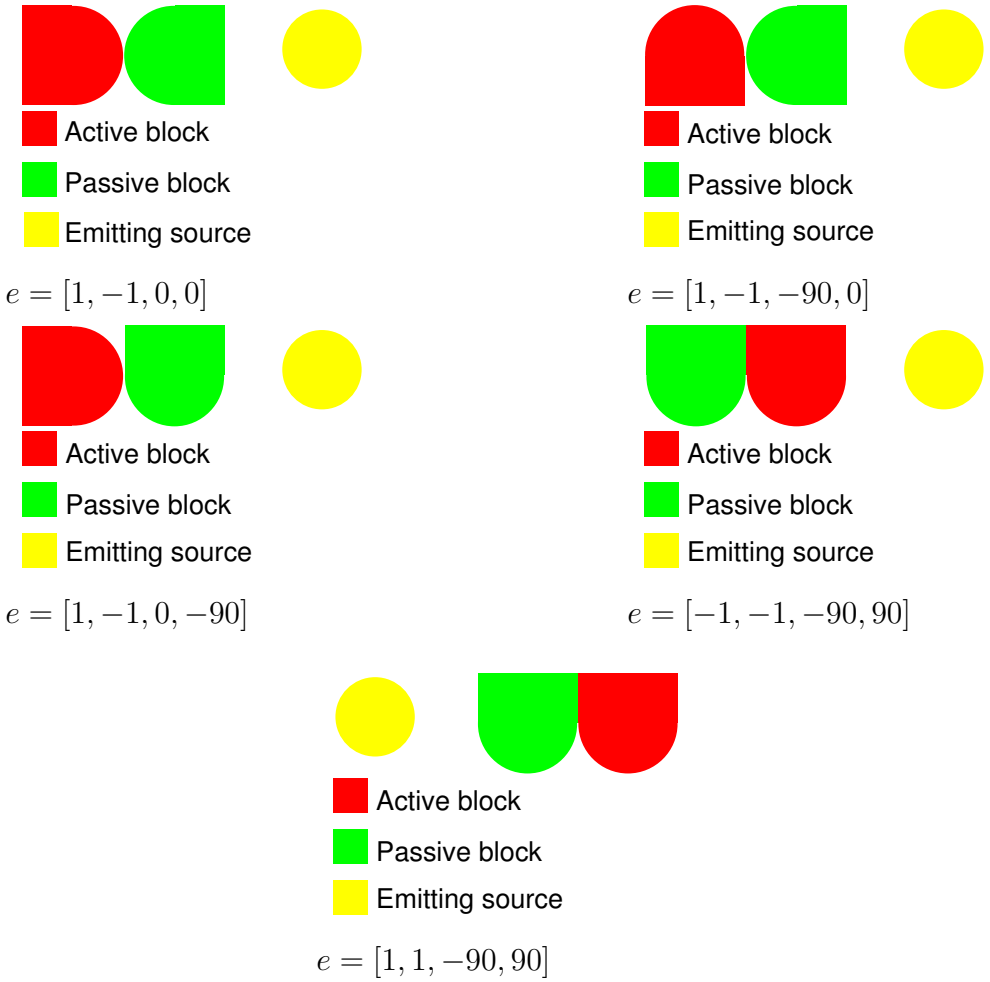

Figure 4.1: Possible states of a robot

It's worth noting that the sensor values don't belong to the learning state. This is because the robot's position (as it's distance to the emitting source) doesn't affect the choice of the action to perform. The robot's position (derived from the sensor values) is used to determine the robot's advance and calculate the reward associated to each performed action.

Learning (actions)

The actions that a robot can perform are defined as a list of movements that will be executed simultaneously. Each movement corresponds to activating one motor. One action can involve one or two movements (since the robot is formed by just one module, the total amount of motors is 2).

According to this, an action is formally described by:

- $a = m_1 \vee a = m_1, m_2$
- $m_i = module|block|angle$
- *module* is the number of the module to which the motor belongs (starting from 0). In this case, since there is only one possible module, this value is always 0. The definition is given in general form, because in later experiments (that involve more than one module) the same definition is used.
- *block* is A if the motor belongs to the active block or P if it belongs to the passive block.
- *angle* is -90 or +90. In the angle definition for actions, the angle is taken to be relative, and not like the angle describing the state of a module which is given as an absolute value. Therefore, in this case 0 is not a valid value.

Given that the actions a robot conformed by only one module can perform are few, they can be explicitly described. According to the values previously indicated, all possible actions that a module can perform are:

- 0|A|-90
- 0|A|+90
- 0|P|-90
- 0|P|+90
- 0|A|-90,0|P|-90
- 0|A|-90,0|P|+90
- 0|A|+90,0|P|-90
- 0|A|+90,0|P|+90

This definition choice for action is because it corresponds to the minimum indispensable data required to modify the robot's configuration (the way the robot has to move around), and therefore it's state. The choice of angles used is due to the fact that these angles (-90, +90) are enough to allow the robot to perform locomotion (even if it's rudimentary) and thus, adding more values would unnecessarily increase the state-action space size.

Learning (reward)

The reward function that was used is defined as:

$$reward(s_t, a_t) = \begin{cases} -5 & \text{if the robot didn't change it's state}^1 \\ distance(s_t, a_t) & \text{otherwise} \end{cases}$$

where $distance(s_t, a_t)$ is the distance travelled during the last action. This value depends on movement the robot did during the last action, but it can be considered to live in the $[-1, 1]$ range.

The penalty given in the case of not changing state is used to disfavour actions that can potentially lead to immobility states.

Using this reward function, the actions that produce a movement in the direction of the emitting source are rewarded in a proportional way to the distance advanced, while the actions that produce a movement in the opposite direction are accordingly penalized.

Results

Initially, we analyzed the impact of the different variables involved in the learning algorithm. Three variables were selected for further study:

- *alpha* and *gamma* are parameters of the *Q-Learning* algorithm, that define the impact that actions have over future decisions
- *epsilon* is a parameter of the policy used to determine the actions to perform. In an ϵ-*greedy* policy, *epsilon* defines the proportion of randomly-chosen actions.

Three series of 90 simulations each were made varying the values of *alpha*, *epsilon* and *gamma*, in order to study which one of them would have the biggest impact on the obtained result.

The variables observed for each simulation were:

[1] in this case, the analyzed state is the real state of the robot, not the state used for learning. In order to determine this fact we compare the robot's state before and after performing an action.

- **advanced distance**

  Distance that the robot moved in the direction of the emitting source.

- **travelled distance**

  Total travelled distance (every movement is taken into account, even those that make it go backwards).

- **executed actions**

  Number of actions the robot executed before reaching it's goal state (with a maximum of 200 actions).

In order to minimize the impact of the result variance, 10 runs were performed with each parameter set, and the results were evaluated using the average of the obtained values.

During these experiments the aim was not set in finding the optimal values for these variables, but to present an overview about the factors that influence the studied problem. Due to this, the performed analysis is not exhaustive and was only used to discover some interesting aspects about the nature of the problem at hand.

By examining the obtained results (that are included in Appendix B), it can be seen that the learning process wasn't successful in those cases where $\epsilon \geq 0.6$, while the cases in which the learning process was faster were when the latter took values lesser than 0.3. These results also suggest that the most convenient values for α and γ are between 0.6 and 0.9.

After this, another three series of 90 simulations each were performed in order to compare the obtained policies. These simulations were performed by just evaluating the policies obtained during the learning phase.

Besides the data analyzed during learning, in order to compare the different policies, we defined a measure of *performance*. This value is calculated according to equation 4.4.

$$performance = \frac{d_{advanced}}{d_{travelled}} \times \frac{k}{n} \tag{4.4}$$

where

$$\begin{aligned} d_{advanced} &= \text{advanced distance} , \\ d_{travelled} &= \text{total travelled distance} , \\ k &= \text{number of required actions}^2 , \\ n &= \text{number of executed actions} . \end{aligned}$$

Examining this equation, it can be seen that the ratio

$$\frac{d_{advanced}}{d_{travelled}}$$

gives a metric about the advance velocity of the robot (i.e. if the performed actions turned out to be mostly in the direction of the emitting source), while the ratio

$$\frac{k}{n}$$

shows the average advance velocity generated by the decisions taken by the robot.

Therefore, the best policies must show a high performance value (close to 1). From this, we can conclude that in order to find optimal policies it's necessary to maximize the value of the performance variable.

The obtained results are shown in table B (included in the Appendix B).

From these results, some graphics can be extracted (figures 4.2, 4.3, 4.4) that explain the variance of performance in relation to each parameter.

Finally, in table 4.1.1, these results are compared with the values obtained by executing an arbitrary policy (manually generated) and a completely random policy.

Policy	Advanced Distance	Travelled Distance	Actions Executed	Performance
Learning[a]	29.50	62.00	56	0.35310
Learning[b]	22.50	65.50	148	0.12850
Random	3.45	153.30	201	0.00391
Arbitrary	29.50	56.70	56	0.35599

[a] The values for the best learned policy are shown

[b] The values for the worst learned policy are shown

Table 4.1: Performance comparison for the learned policies versus the control policies

In this table it can appreciated that all the obtained policies performed better than the random policy, and even though they weren't as performing as the manually constructed policy, this is because due to the simplicity (and

[2] we consider this value to be the minimum amount of actions required to reach the goal state, i.e. the amount of actions performed by the optimal policy

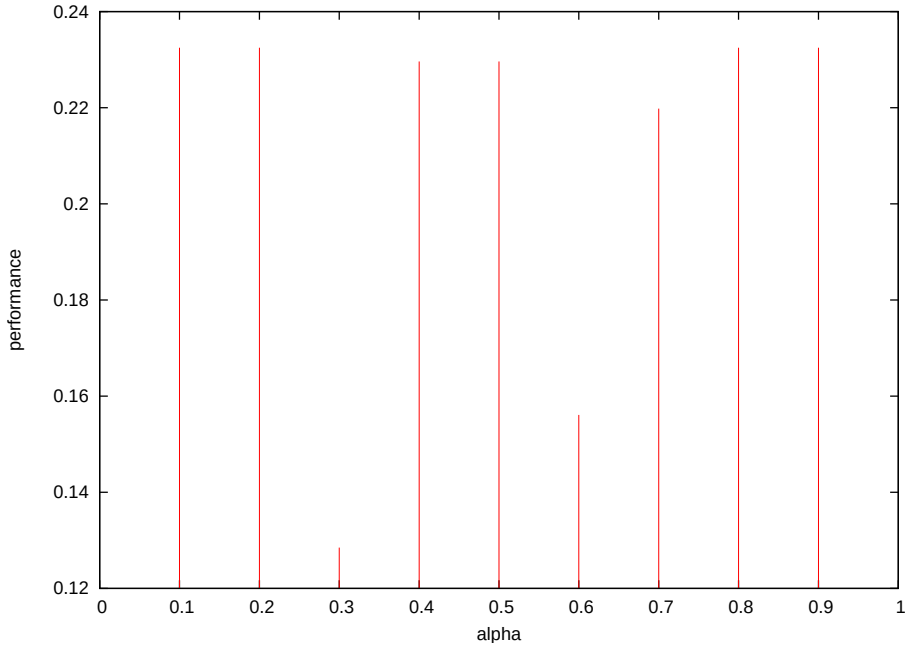

Figure 4.2: Performance of the policies obtained in relation to the parameter α, for $\epsilon = \gamma = 0.5$.

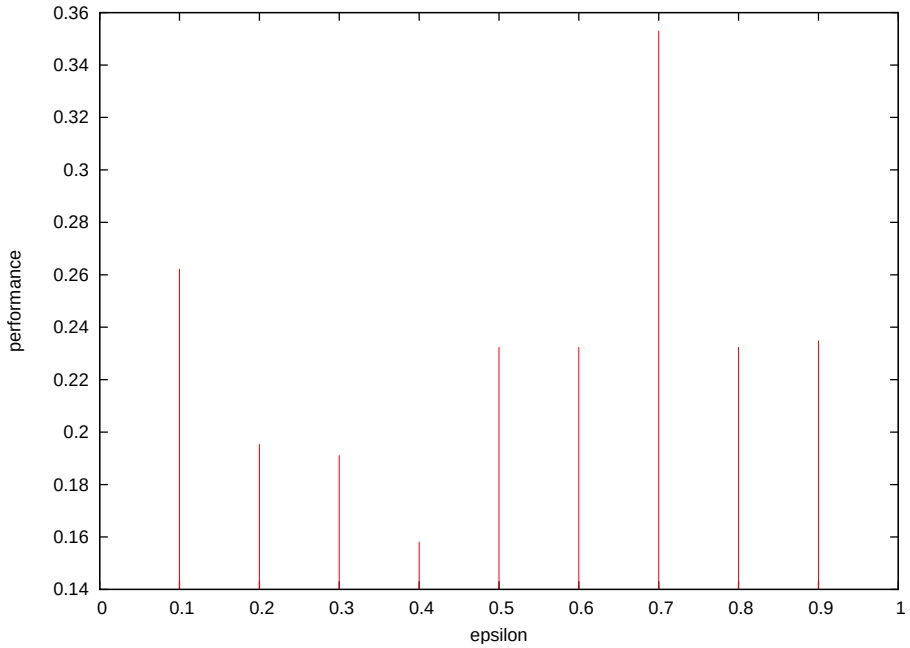

Figure 4.3: Performance of the policies obtained in relation to the parameter ϵ, for $\alpha = \gamma = 0.5$.

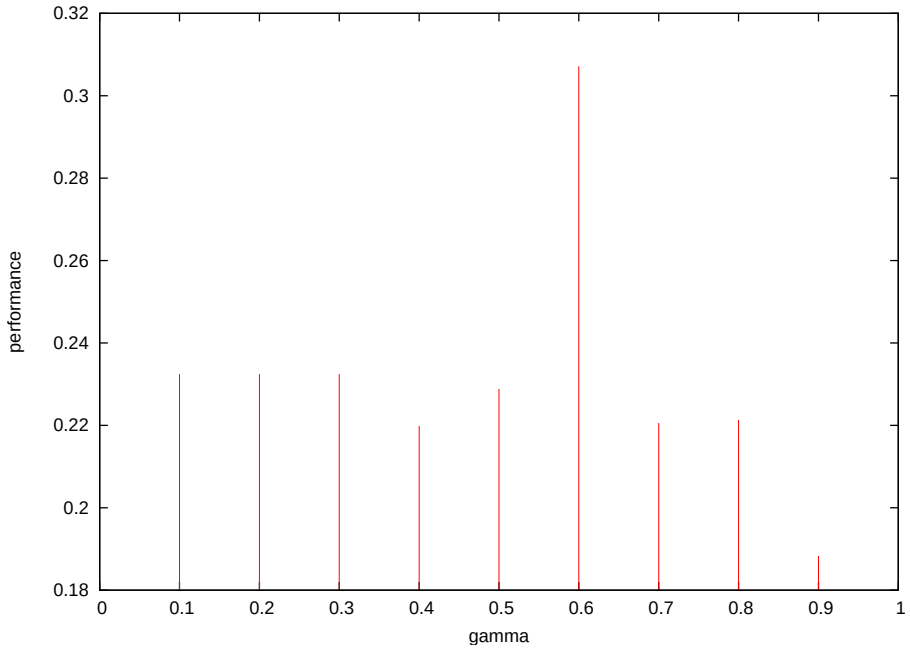

Figure 4.4: Performance of the policies obtained in relation to the parameter γ, for $\alpha = \epsilon = 0.5$.

thus it's reduced size) of the state and action space, it is easy to manually determine an optimal policy. However, in the case that the robot would consist of more modules and/or could perform a greater number of actions, finding an optimal policy manually wouldn't be as easy. In those circumstances, having a learning algorithm that provides close to optimal policies would be highly esteemed.

By examining the obtained results, it's possible to see that the optimal values for α are at the extremes of the spectrum. In the case of using lower-end values the impact the actions performed would have over the learning process would be minimized (in an extreme case, $\alpha = 0$ would mean no learning at all), while by using values in the upper-end of the spectrum would mean to base the decisions almost entirely on next state-action pair (i.e. $Q(s', a')$) estimation in order to determine the next action to execute; this is somehow related to not learning, since a high value for α would make the robot "forget what it learned at each time step".

The value for γ suggested by the results is expectable, since it represents a balance between using recent experience and "forgetting" long-ago acquired experience (would that not be the case, experience would not be taking into account at all – γ too low – or "mistakes would not be forgotten" – γ too high).

The value obtained for the initial ϵ is however interesting. The best results were obtained when ϵ showed a relatively high value ($\epsilon = 0.7$). However, in many case in which ϵ had a value in it's upper-end range ($\epsilon > 0.6$), the robot couldn't reach the goal state during it's learning phase. This means that even though the best policies involve a high degree of exploration, which actually makes sense, since it's *necessary* to make a certain amount of exploration in order to learn an optimal policy, a high proportion of exploration can also drive to underuse the acquired experience, thus producing a suboptimal performance. The fact that the best policies found showed a high value for this variable is due to the bounded size of the state space, which allows for an *initial* high exploration to rapidly find the best actions for each state (in the performed simulations, exploration is strong at the beginning, but then decreases as time goes by).

Conclusions

In the light of the obtained results, it can be verified that it is *feasible* to perform learning tasks (particularly using Reinforcement Learning) with Self-Reconfigurable Modular Robots like the M-TRAN robot.

It was shown that the developed simulator fulfills the necessary requirements to be able to perform this kind of experiments, and that like in other similar experiments it's necessary to carefully define the learning algorithm related aspects (state, action, reward function).

The obtained results are consistent with the existing knowledge about Reinforcement Learning. That is, in regard to the studied problem, the values for the analyzed variables that produced the best results, are coherent.

Finally, even though the proposed locomotion task was simple, the success obtained from the performed simulations allows us to suggest that using the Reinforcement Learning technique for the locomotion task in M-TRAN Self-Reconfigurable Modular Robots has a very real and important utility.

4.2 Reconfiguration

In this case we tried to verify an M-TRAN's robot capacity for self-reconfiguration in order to obtain behaviours for the reconfiguration task using Reinforcement Learning.

Next we will describe the performed experiments.

4.2.1 Experiment 2: Basic Reconfiguration

In this experiment we worked with a robot composed of three modules. The initial configuration was linear, meaning that all modules where aligned as is shown in figure 4.5.

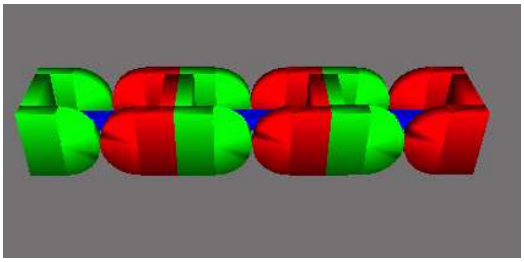

Figure 4.5: Linear configuration for a three module M-TRAN robot, corresponding to the initial state for the experiment number 2

The goal of this experiment is that the robot achieves the configuration shown in figure 4.6.

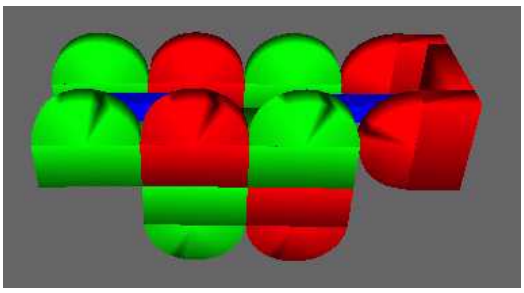

Figure 4.6: Final configuration for a three module M-TRAN robot for the experiment number 2

The robot's state is represented by a list containing each module's state. Likewise, a module's state is represented by a pair of angles, corresponding to the module's motors, that is

$$\begin{aligned} state &= [state_{\text{module}_1}, state_{\text{module}_2}, state_{\text{module}_3}] \ , \\ state_{\text{module}_i} &= \text{'}\alpha_i::\beta_i\text{'} \ . \end{aligned} \tag{4.5}$$

According to this representation, the initial and final states of the experiment are

$$\begin{aligned} state_{initial} &= [\text{'0.0::0.0'}, \text{'0.0::0.0'}, \text{'0.0::0.0'}] \\ state_{final} &= [\text{'0.0::-90.0'}, \text{'-90.0::90.0'}, \text{'90.0::-90.0'}] \end{aligned}$$

The actions the robot can perform are only those given by the angles of the different motors. These actions are defined by

$$\begin{aligned} action &= \text{'}module|block|angle\text{'} , \\ module &= 0 \vee 1 \vee 2 , \\ block &= A \vee P , \\ angle &= \text{-}90 \vee +90 . \end{aligned} \tag{4.6}$$

The actions defined by this equation are called "simple actions", because each action allows to modify the angle of only one motor. In future experiments, "complex" actions were also used to modify more than one motor at the same time.

The used reward function is defined as:

$$reward = \begin{cases} 1 & \text{if the final state is reached} \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

Results

After performing 13 learning episodes, we obtained a policy that solved the proposed task. As can be seen in figure 4.7, during the learning phase, the number of performed actions in each episode tends to diminish. This is because the robot is using the acquired experience (also meaning, it learns). However, it can also be seen that in certain episodes the robot needs more actions than in previous episodes to reach the goal state. Even though this might suggest that the robot is not using previous experience, this is actually due to some exploration being done every now and then. Thanks to this exploration phases the robot is able to determine which states are going to lead to the final state. At the same time, this makes it have to try out more actions in order to reach those states.

Another aspect that can be observed in this figure is a curve named "global tendency", that is calculated as a linear approximation of the data, using the least mean squares method.

In figure 4.8 it can be seen that by using the obtained policy, the same amount of actions are always required. This shows that the policy is stable,

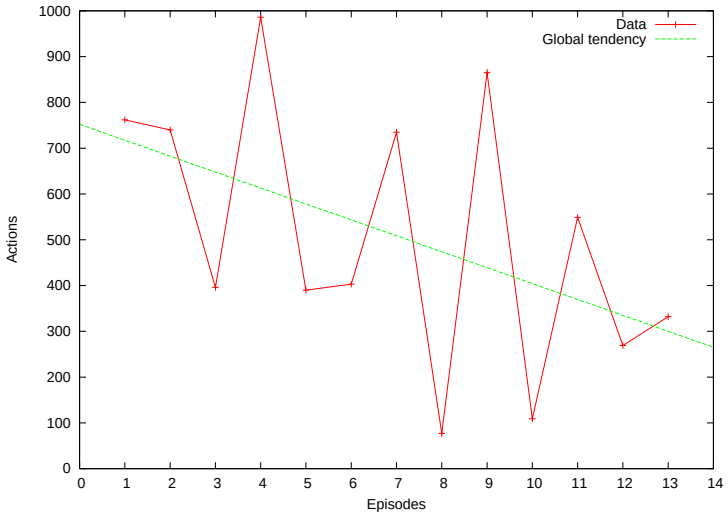

Figure 4.7: Number of actions executed before reaching the final state during the learning phase, in order to obtain a reconfiguration policy during experiment number 2

meaning that in every validation instance, the robot behaves in the same way. Another noteworthy aspect is the difference between the number of actions performed during the learning phase and during the evaluation of the learned policy. It's noteworthy that even though a considerable amount of actions is necessary during the learning phase (between 100 and 1000 actions were executed), when evaluating the learned policy just 5 actions are needed to reach the final state. The difference between those values shows that after performing a number of episodes enough to determine a stable policy, the latter shows to be highly efficient.

Discussion

As the obtained results show, the first attempt to learn a policy for solving a reconfiguration task was successful. Even though the proposed task was a simple one (the target configuration was easily reachable), this first success motivates further and more complex experiments.

4.2.2 Experiment 3: From linear to circular configuration, using simple actions

Once we verified that a simple reconfiguration task was possible to solve using Reinforcement Learning, we modified the target configuration in order

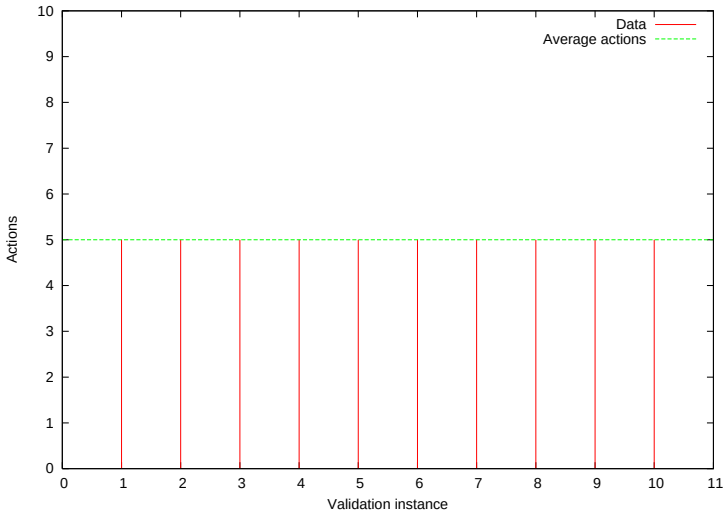

Figure 4.8: Number of actions executed before reaching the final state, using the learned policy during experiment number 2

to test this technique with a different configuration. In this case, we started from the same initial configuration as in the experiment 2, and tried to reach a "circular" configuration, as shown in figure 4.9.

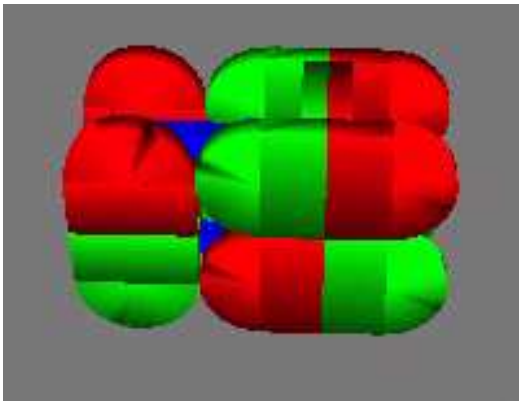

Figure 4.9: Final circular configuration for a 3-module robot, used during experiment 3

According to the state specification given for experiment 2 (which remains the same in this experiment), the target configuration, as shown in figure 4.9 can be expressed as

$$state_{final} = [\text{'-90.0::90.0'}, \text{'0.0::90.0'}, \text{'-90.0::0.0'}]$$

In this experiment we used the same definition for actions and the same reward function as in experiment 2.

Results

13 learning episodes were performed, at the end of which we evaluated the learned policy. The results of these first 13 episodes are shown in figures 4.10 and 4.11.

In the case of figure 4.10, the amount of performed actions before reaching the target configuration is shown. It's worth noting that the global tendency is decreasing, which like in the previous experiment, indicates that the robot is learning. The increase peaks are due to the exploration phases (as in the previous case). Another thing to note is that the greater complexity of the target configuration causes a greater number of actions to be executed (on average) in order to reach the target configuration than during experiment 2.

In figure 4.11 the amount of actions needed to reach the target configuration is shown according to the learned policy. In this case, it can be appreciated that the obtained policy is still unstable (in the sense that it doesn't always determines the same actions). This is because there hasn't been enough exploration in order to determine the optimal action sequence. Therefore, in some states more than one action have a maximal Q value associated, which produces that it's not possible to deterministically select the optimal action for those states.

Because of this, more learning episodes were performed. Learning and evaluation episodes were performed alternately until a stable policy was found (meaning that it executes always the same actions).

Results for these episodes are shown in figures 4.12 - 4.15.

In figures 4.12 and 4.14 it can be seen how the amount of required actions decreases as more learning episodes are performed, finally converging to a stable value. This indicates that the robot is learning, effectively using previous experience.

On the other hand, in figure 4.13 it can be seen that after 17 episodes a stable policy was still not found, but that, according to figure 4.15, after 19 episodes a stable and highly efficient (regarding the difference between the number of actions performed during learning and evaluation) policy is effectively found.

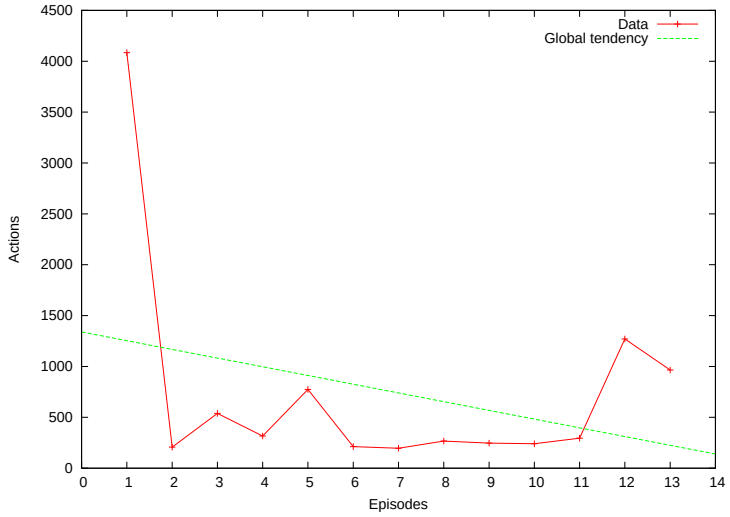

Figure 4.10: Number of actions performed before reaching the target configuration, during the learning phase (13 episodes) for a reconfiguration task policy for experiment 3

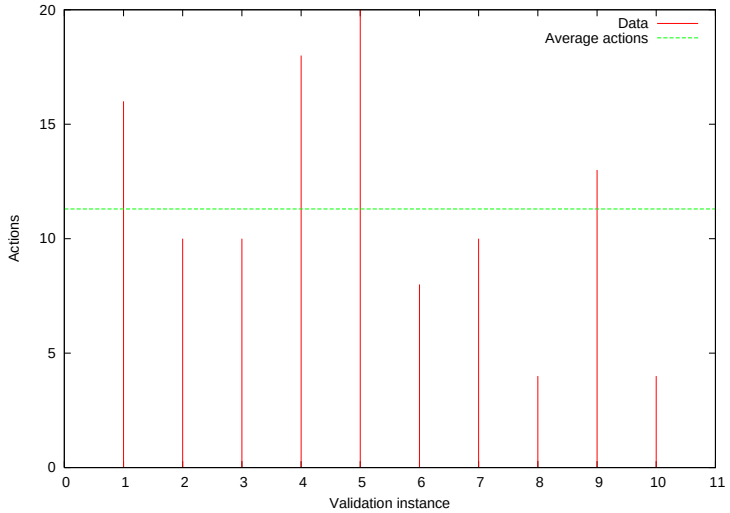

Figure 4.11: Number of actions performed before reaching the target configuration, using the learned policy (after 13 episodes) for experiment 3

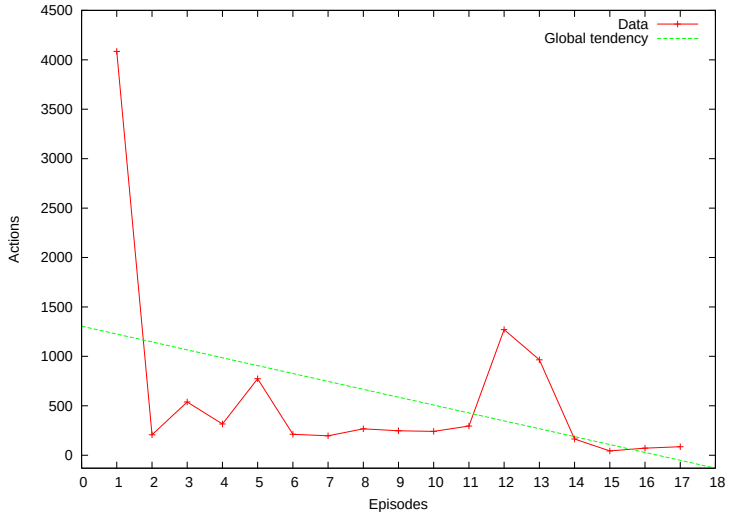

Figure 4.12: Number of actions executed before reaching the target configuration, during the learning phase (17 episodes) of a reconfiguration task policy for experiment 3

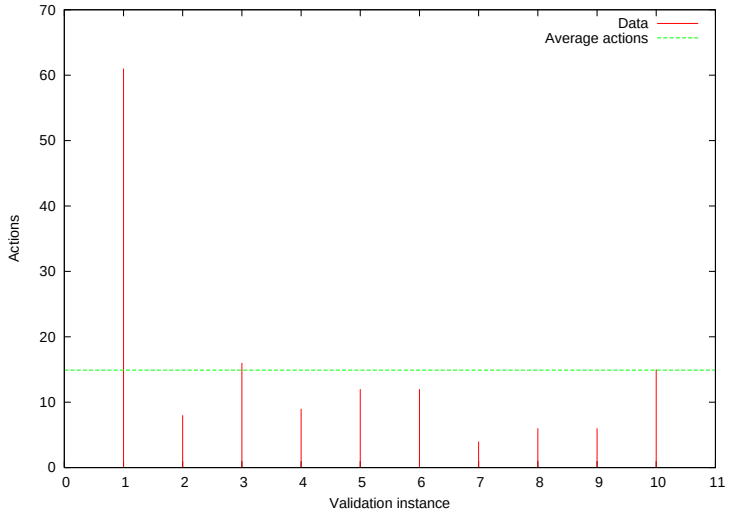

Figure 4.13: Number of actions executed before reaching the target configuration, using the learned policy (after 17 episodes) for experiment 3

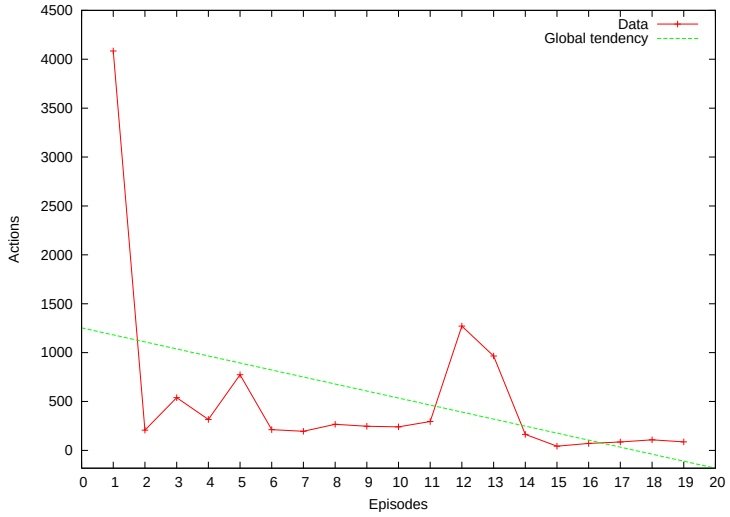

Figure 4.14: Number of actions executed before reaching the target configuration, during the learning phase (19 episodes) of a reconfiguration task policy for experiment 3

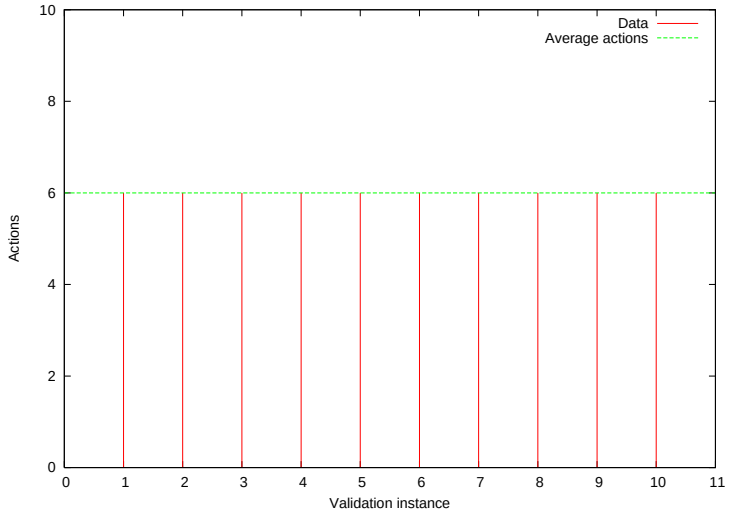

Figure 4.15: Number of actions executed before reaching the target configuration, using the learned policy (after 19 episodes) for experiment 3

Discussion

As can be observed, the fact that as more learning episodes are performed the number of actions executed before reaching the target configuration decreases (due to learning) still holds. Likewise, it can be observed how after a certain amount of episodes, the learned policy stabilizes on a minimal required number of actions.

4.2.3 Experiment 4: From linear to circular structure, using complex actions

In this case, we introduce a variation respect the previous experiment. Instead of using "simple" actions, we redefined the possible actions so that several motors can be simultaneously activated. The main reason for this change is to evaluate if by being able to simultaneously activate several motors the final policy results more efficient.

"Complex" actions are thus defined as

$$\begin{aligned} action &= 'movement' \vee 'movement; action'^{3} \ , \\ movement &= \text{simple action, according to equation 4.6 .} \end{aligned} \tag{4.8}$$

This definition is complemented by two restrictions:

1. Within an action, each movement must refer to a different motor
2. There can be a maximum of 6 movements in a complex action (or generally speaking, the number of motors present in the robot). This restriction comes from the first one.

Results

In this experiment, we evaluated the learned policy after a few episodes (because we also tried to verify if using complex actions would make the learning process faster).

The results can be seen in figures 4.16 - 4.21.

[3] even though an action is defined as a sequence of movements, the order in which those movements are given doesn't matter; all actions that are permutations of the same movements are considered to be the same action

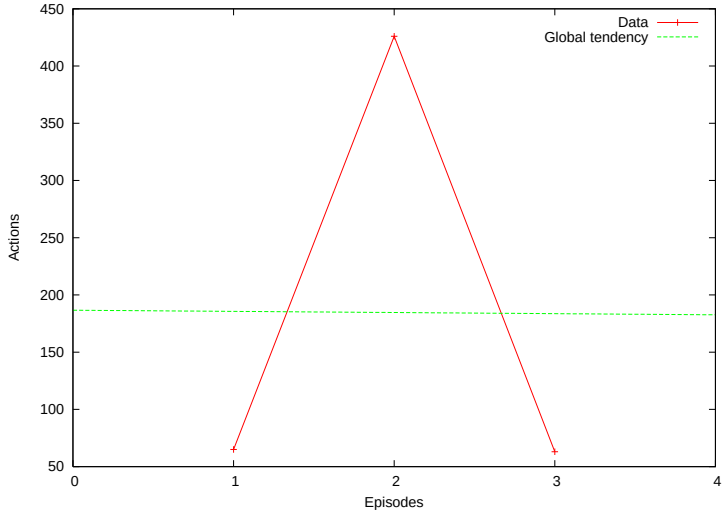

Figure 4.16: Number of actions performed before reaching the target configuration, during the learning phase (3 episodes) of a reconfiguration task policy for experiment 4

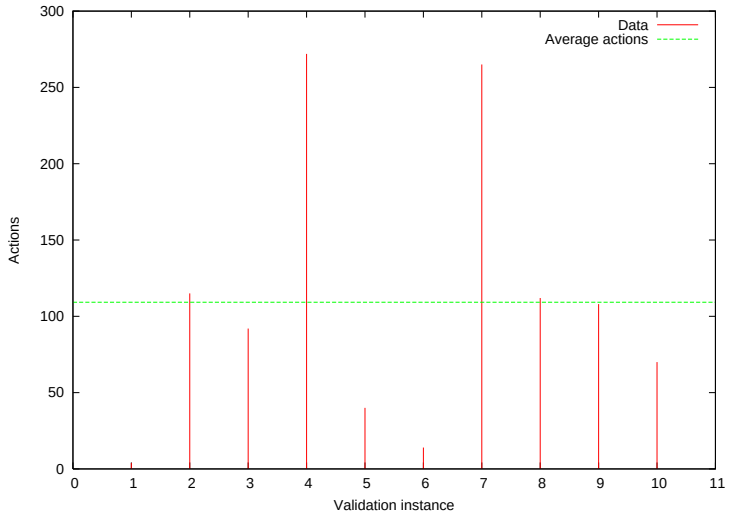

Figure 4.17: Number of actions performed before reaching the target configuration, using the learned policy after 3 episodes, for experiment 4

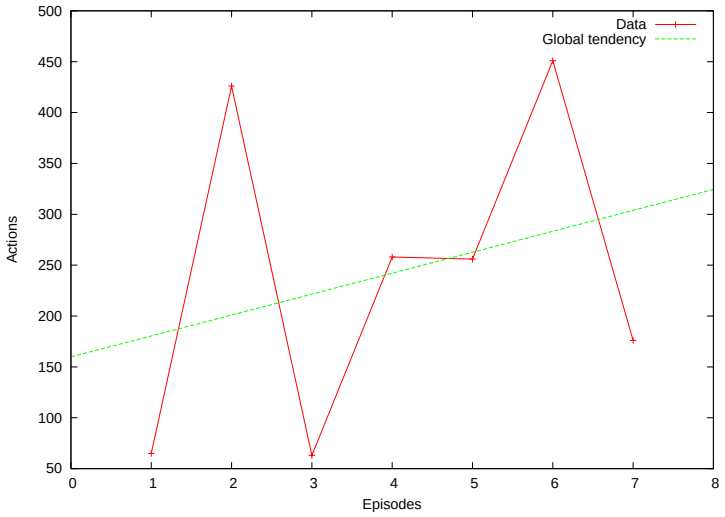

Figure 4.18: Number of actions performed before reaching the target configuration, during the learning phase (7 episodes) of a reconfiguration task policy from experiment 4

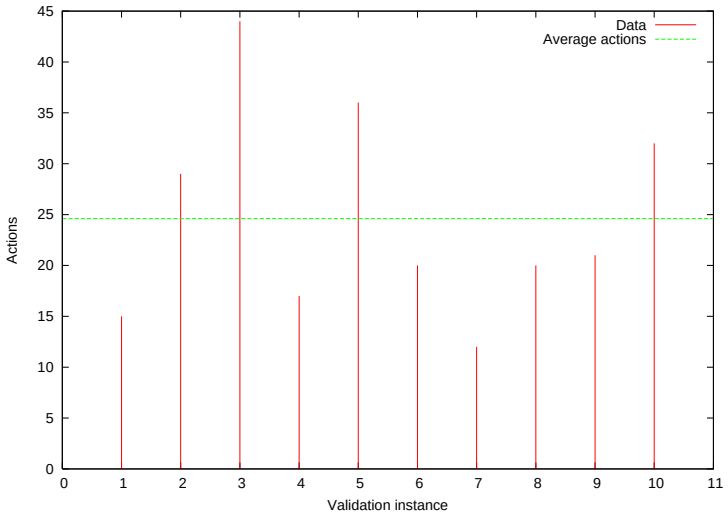

Figure 4.19: Number of actions performed before reaching the target configuration, using the learned policy after 7 episodes, for experiment 4

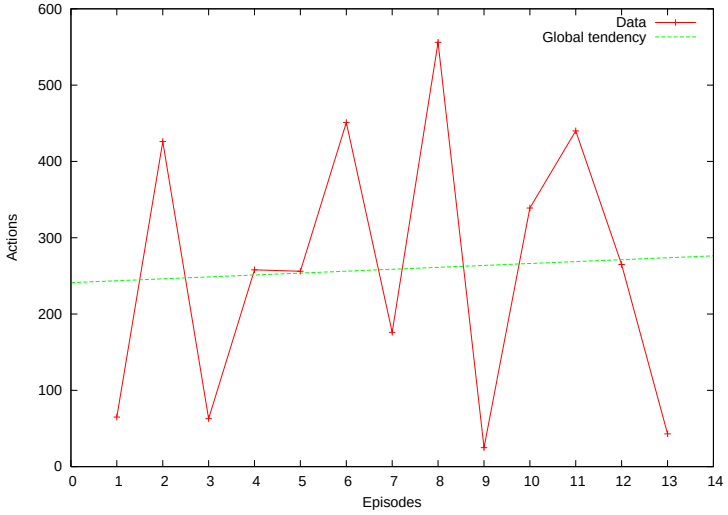

Figure 4.20: Number of actions performed before reaching the target configuration, during the learning phase (13 episodes) of a reconfiguration task policy, for experiment 4

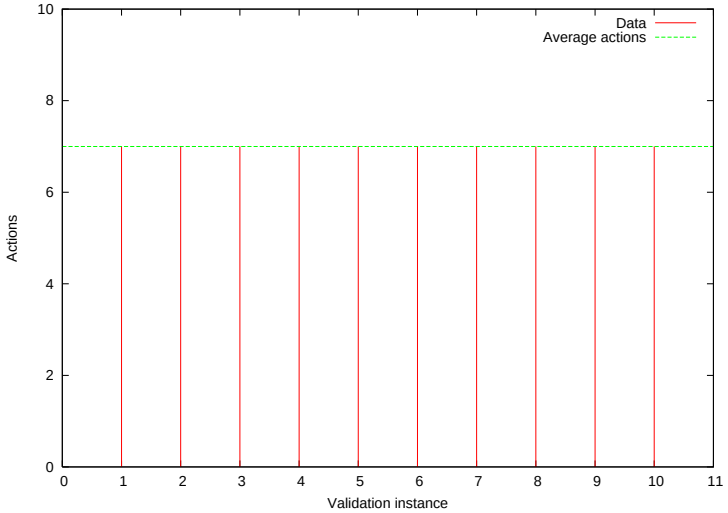

Figure 4.21: Number of actions performed before reaching the target configuration, using the learned policy after 13 episodes, for experiment 4

Discussion

In the first figure it can be seen that the number of actions performed during the learning phase fluctuates greatly. While this alone does not indicate that the policy hasn't yet been refined enough, if we also take into account the next figure (4.17) it can be verified that that is exactly what is happening (since the obtained policy is unstable). The reason for that is that the number of learning episodes performed is too little. In the next figure (corresponding to the learning phases for 7 episodes), the increasing global tendency still holds. However, the number of performed actions is on average, much lower than during the first learning episodes on previous experiments. This fact indicates that the learning process is effectively being accelerated by using complex actions. The obtained policy is however, still unstable (as can be deduced from figure 4.19), which indicates that more learning episodes still have to be performed.

Finally, after 13 learning episodes, a stable policy is obtained (see figure 4.21). Even though during the learning phase the global tendency is increasing, the increase rate is notably lower than to that shown in figure 4.18, which demonstrates that on average term the amount of actions performed during the latter episodes decreases.

Another factor worth noting is that the amount of actions executed by the policy decreases (on average) until a stable policy is found (which always executes the same amount of actions).

Even though the number of actions required by the obtained policy is not strictly lower than the number of actions required by the policy obtained in experiment 3, and even though it cannot be affirmed that the former is better (or more efficient) than the latter, the obtained result is good enough, since the number of episodes required to reach a stable policy is less than in the case of experiment 3, and the difference between the number of actions those two policies perform is minimal.

4.2.4 Experiment 5: From linear to circular structure, with docking/undocking and using complex actions

Having already succeeded in solving a basic reconfiguration task, in this experiment we try to complicate the situation by introducing a yet unexplored aspect of the robot's capacities: coupling between modules. The goal of this experiment is to reach a similar target configuration than the one used in the previous two cases (i.e. starting from a linear configuration, trying to reach a circular configuration), with the difference that in this case, the robot will

have to end with all it's modules attached together, forming a ring.

To achieve this it's necessary to:

1. Modify the definition of state, to include coupling between modules
2. Modify the definition of action, to allow for modules to dock and undock

In order to be able to introduce changes that affect the couplings between different modules, and learn from this, it's necessary that the state used for learning describes those couplings. We thus redefine the state as

$$\begin{aligned} state &= (angles, couplings)\ , \\ angles &= \text{robot state according to equation 4.5}\ , \\ couplings &= [coupling_0, coupling_1, \ldots]^{4}\ , \\ coupling_i &= (module_a, module_b, side^{5})\ , \\ side &= \text{'bottom'} \vee \text{'front'} \vee \text{'rear'}\ . \end{aligned} \tag{4.9}$$

According to this new state definition, the initial and final states for this experiment are:

$$\begin{aligned} state_{initial} &= (['0.0::0.0', '0.0::0.0', '0.0::0.0'], \\ &\quad [(1, 0, \text{'bottom'}), (2, 1, \text{'bottom'})]) \\ state_{final} &= (['\text{-}90.0::90.0', '0.0::90.0', '\text{-}90.0::0.0'], \\ &\quad [(1, 0, \text{'bottom'}), (2, 1, \text{'bottom'}), (0, 2, \text{'bottom'})]) \end{aligned}$$

To allow docking and undocking between modules, the available actions are defined as

$$action' = complex\text{-}action \vee coupling\ , \tag{4.10}$$

$$\begin{aligned} coupling &= \text{'dock}|module_a|module_b|side^{7}\text{'} \vee \\ &\quad \text{'undock}|module_a||side^{7}\text{'}\ , \end{aligned} \tag{4.11}$$

$$complex\text{-}action = \text{action as defined by equation 4.8}\ .$$

By using this new definitions for state and action, it is possible that during an episode the robot falls into a state in which it will be practically

[9] all coupling permutations are considered as the same. See note regarding to permutations in the definition of complex actions (equation 4.8)

[5] of the a module

impossible for it to reach the target configuration. For example, if a module gets detached, while theoretically it would be possible for it to reattach to the robot, the conditions that must hold for this to happen are so strict that the probability for it is quite low.

For this reason, if the robot should find itself in such a situation, instead of allowing it to continue, we opted for aborting the episode, returning a negative reward. The reward function used for this experiment is given by the following equation

$$reward = \begin{cases} 1 & \text{if the target configuration is reached} \\ -1 & \text{if some module got detached} \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

Results

43 learning episodes were performed, and every one had to be aborted, without reaching the target configuration.

Figure 4.22 shows the amount of actions executed during each learning episode.

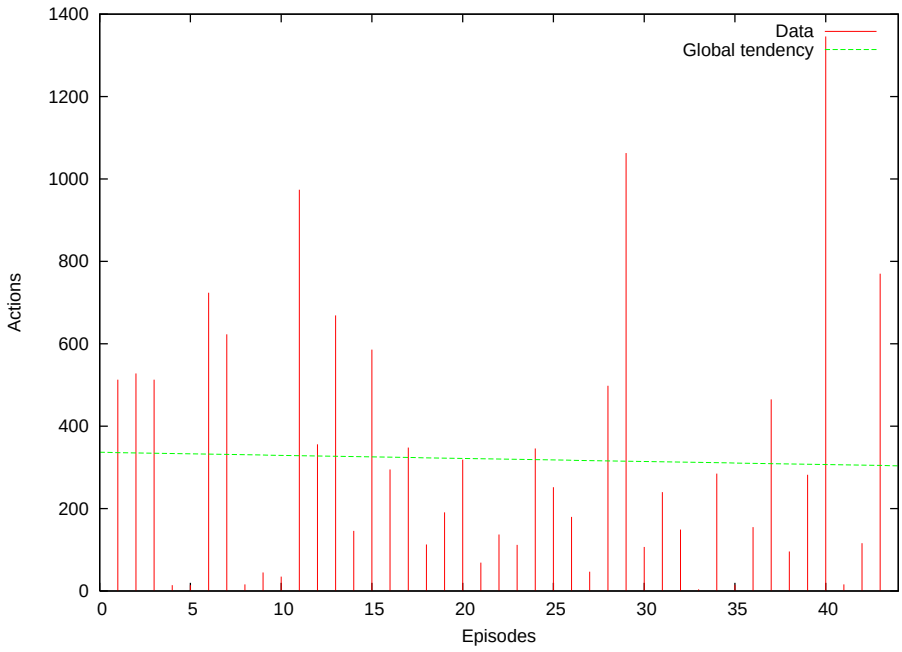

Figure 4.22: Number of actions performed during each learning episode, before being aborted.

Discussion

Having introduced the possibility of aborting an episode where the robot hasn't reached the target configuration makes it necessary to perform many more episodes, or to reevaluate the definition of episode.

From what can be seen in figure 4.22, there is a decreasing tendency in the number of actions performed during each episode, before it gets aborted. While this tendency might indicate that effectively more episodes are needed in order to reach a stable policy, since every episode was aborted, this correlation might not be true. This comes from the fact that the reward function assigns rewards at the end of each episode. Therefore, the only information that can be extracted from an aborted episode is that the series of actions taken during that episode are not effective for reaching the target configuration.

4.2.5 Experiment 6: From linear to circular structure, with docking, no undocking, using simple actions

Because of the results obtained during experiment 5, we modified the definitions of action and reward function, in order to avoid the encountered problems.

In the case of the available actions, we used a similar definition to that given by equation 4.10, but instead of using complex actions, in this case only simple actions were allowed, and we restricted the possibility of coupling actions only to the *dock* action. That means, that we basically simplified the available actions and avoided the possibility of detaching a module in order to avoid having to abort episodes. This decision was based on the fact that it was not necessary to undock any module in order to reach the target configuration.

The reward function was defined as

$$reward = \begin{cases} -1 & \text{if the episode was aborted } ^{6} \\ partialReward & \text{otherwise} \end{cases} \tag{4.13}$$

where *partialReward* is the function defined in table 4.2.

According to this definition, the reward function assigns rewards for reaching the target angles configuration and for reaching the target couplings con-

[6] even though we just mentioned that the actions were modified in order to avoid having to abort episodes, having this case contemplated by the reward function doesn't produce any negative impact, and prevents unforeseen cases in which the episode has to be aborted.

```
def partialReward(state, target):
     angles, couplings = state
     reward = 0
     if angles == target[0]:
         reward += 1
     if couplings == target[1]:
         reward += 1
     return reward
```

Table 4.2: Definition of function *partialReward*.

figuration, having a maximal value when both parts of the state are correct (i.e. when the target configuration has been reached).

Results

The results obtained from the simulations are shown in figures 4.23 - 4.28.

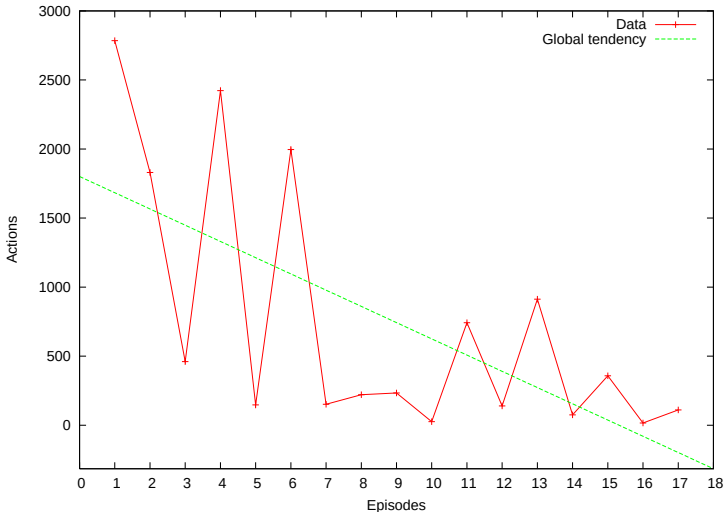

Figure 4.23: Number of actions performed before reaching the target configuration, during the learning phase (17 episodes) of a reconfiguration task policy, for experiment 6

Discussion

In figures 4.23, 4.25 and 4.27 the evolution of the number of actions performed during the learning phase can be observed, as more episodes were performed.

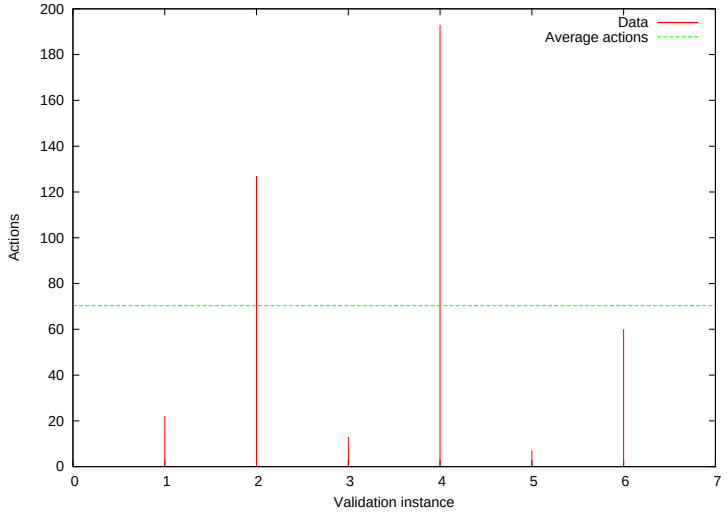

Figure 4.24: Number of actions performed before reaching the target configuration, using the learned policy after 17 episodes, for experiment 6

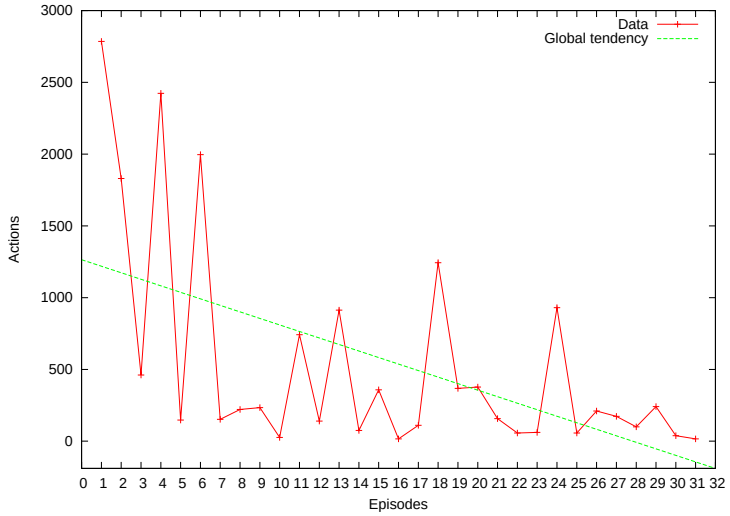

Figure 4.25: Number of actions performed before reaching the target configuration, during the learning phase (31 episodes) of a reconfiguration task policy, for experiment 6

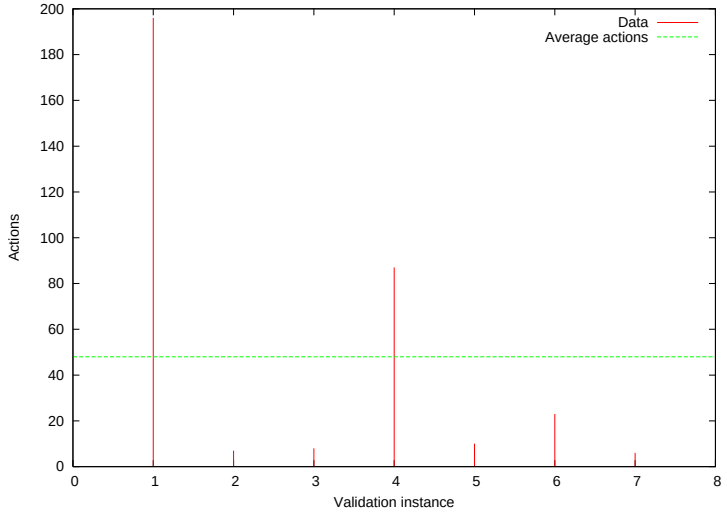

Figure 4.26: Number of actions performed before reaching the target configuration, using the learned policy after 31 episodes, for experiment 6

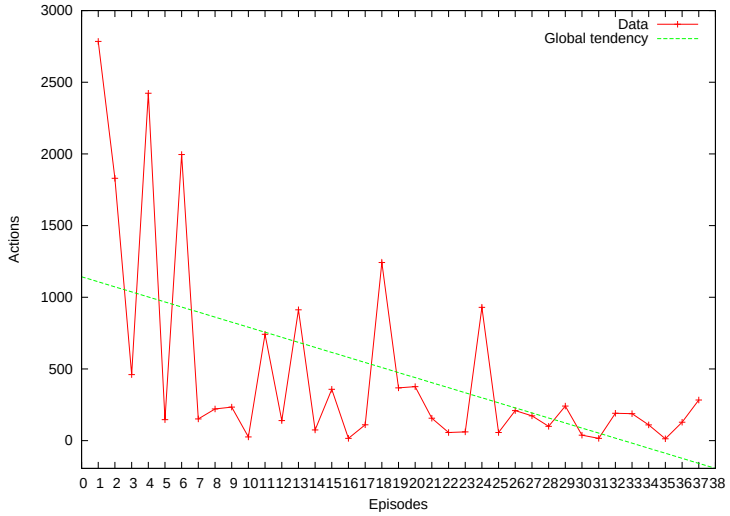

Figure 4.27: Number of actions performed before reaching the target configuration, during the learning phase (37 episodes) of a reconfiguration task policy, for experiment 6

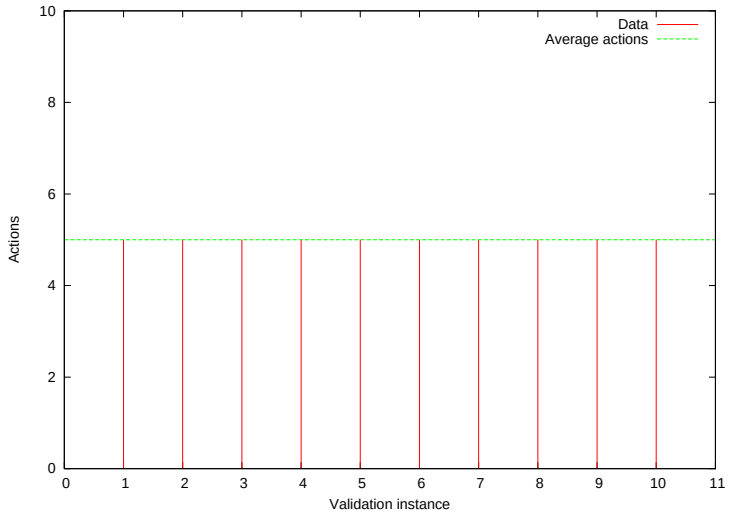

Figure 4.28: Number of actions performed before reaching the target configuration, using the learned policy after 37 episodes, for experiment 6

The negative tendency indicates the effect of the learning process, based on the experience gathered from previous episodes. However, that tendency tends to be less decreasing as more episodes are performed. This indicates that the number of actions performed during those episodes tend to stabilize, which suggests a greater usage of the gathered experience, represented by the obtained policy. This means that if the number of actions performed during the learning phase stabilizes, it's highly probable that a stable policy is being reached. The difference between the number of actions performed during the first learning episodes and the number of actions performed during the last episodes shows a great improvement, which is due to the fact that the robot has learned.

In figures 4.24, 4.26 and 4.28 the evolution of the performance of the policy derived from the learning phase can be seen as more learning episodes were done. Initially, the policy turned out to be highly unstable, while in the latest stages, while it still wasn't stable, the number of actions performed on average by the policy were notably inferior than in the early learning stages.

Finally, figure 4.28 shows that a stable policy is reached, and that it involves a very little number of actions to reach the target configuration. Clearly this problem is more complex than those introduced in experiments 3 and 4, which reflects in the greater number of episodes that had to be performed (37 learning episodes were necessary to reach a stable policy).

4.2.6 Experiment 7: From circular to linear structure, with docking/undocking, using complex actions

Having completed the first phase of experimentation (where we studied different variations of the same problem – starting from a linear configuration and trying to reach a circular configuration), the next two experiments deal with the opposite task, that is, starting from a circular and fully coupled configuration, and trying to reach a linear configuration.

The actions and state defined for this experiment are the same as those used for experiment 5.

The reward function, however, had to be modified. Since in this experiment it is necessary to detach modules, it might be possible that some module would be completely decoupled from the rest of the robot, and while it theoretically could reattach itself, this is so improbable according to the conditions that must hold, that it resulted more practical to assign a negative reward (so as to avoid this from happening in the future), and aborting the simulation (like it was done in experiment 5).

For this reason, the reward function was defined as

$$reward = \begin{cases} 1 & \text{if the target configuration is reached} \\ -1 & \text{if the simulation was aborted} \\ 0 & \text{otherwise} \end{cases} \tag{4.14}$$

Since it's possible to abort simulations, we redefined an episode as the sequence of simulations necessary to reach the target configuration (i.e. the sequence of aborted simulations plus the simulation that returns a positive reward). According to this, the number of actions performed during an episode is the sum of the number of actions performed during each simulation belonging to that episode.

Results

In this experiment 13 episodes were performed, which represent a total of 70 simulations. At the end of those 13 episodes no policy that would fulfil the goal was reached (the obtained policy after 13 episodes cycles between 3 configurations all of which are completely coupled).

This cyclic behaviour can be observed in figure 4.29, which represents a full cycle produced by the actions selected by the learned policy.

The simulation results are shown in figures 4.30 - 4.33.

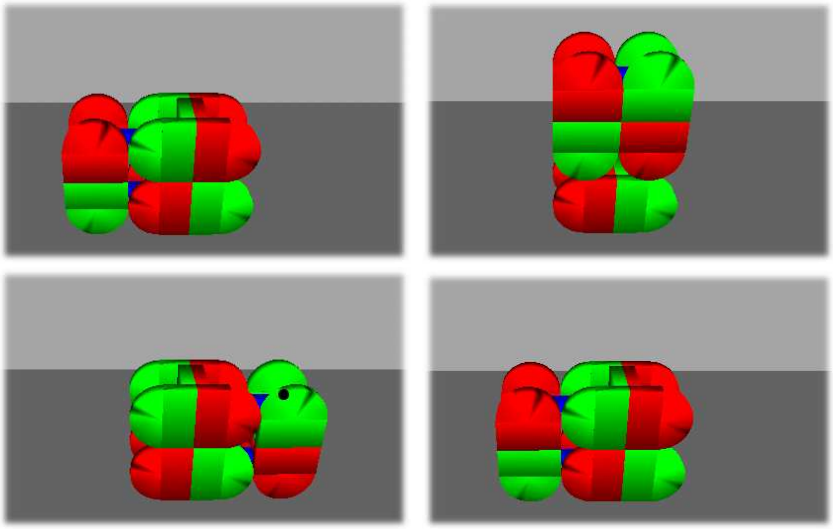

Figure 4.29: States corresponding to a cycle in the actions produced by the learned policy for experiment 7. Top left: state 1. Top right: state 2. Bottom left: state 3. Bottom right: state 4 (equal to state 1).

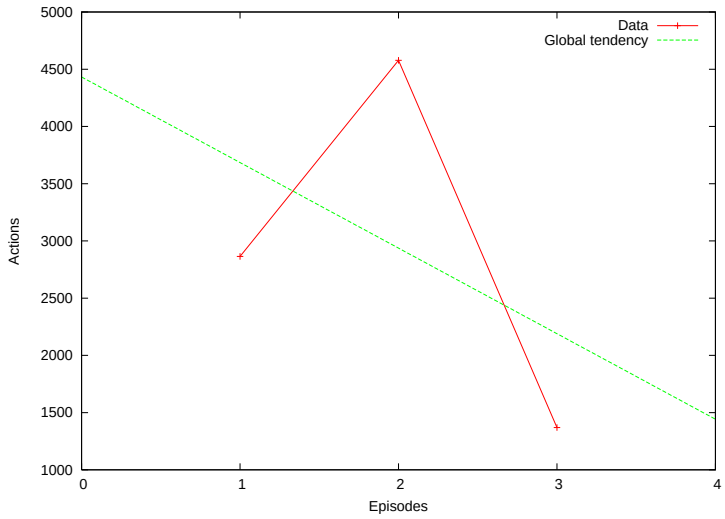

Figure 4.30: Number of actions performed before reaching the target configuration, during the learning phase (3 episodes) of a reconfiguration task policy, for experiment 7

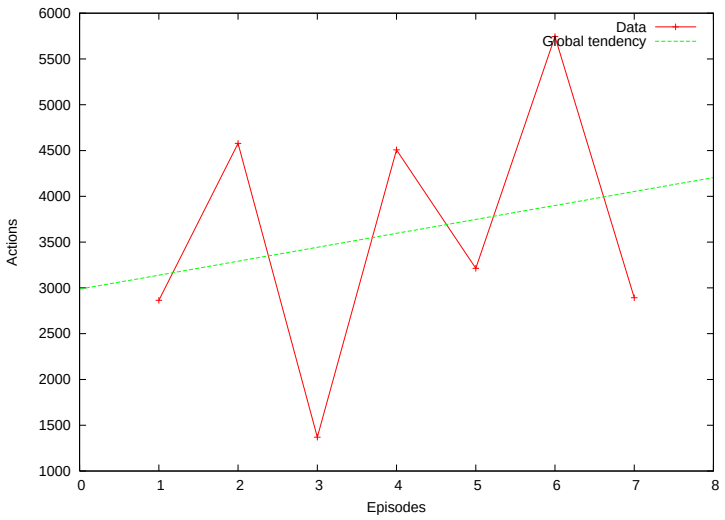

Figure 4.31: Number of actions performed before reaching the target configuration, during the learning phase (7 episodes) of a reconfiguration task policy, for experiment 7

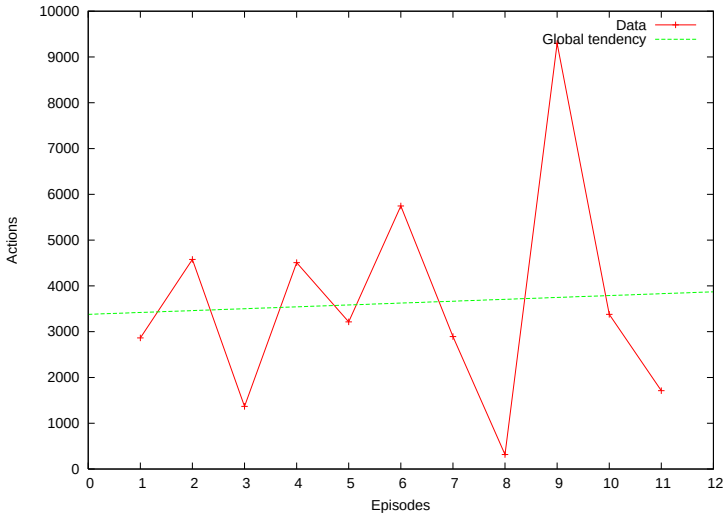

Figure 4.32: Number of actions performed before reaching the target configuration, during the learning phase (11 episodes) of a reconfiguration task policy, for experiment 7

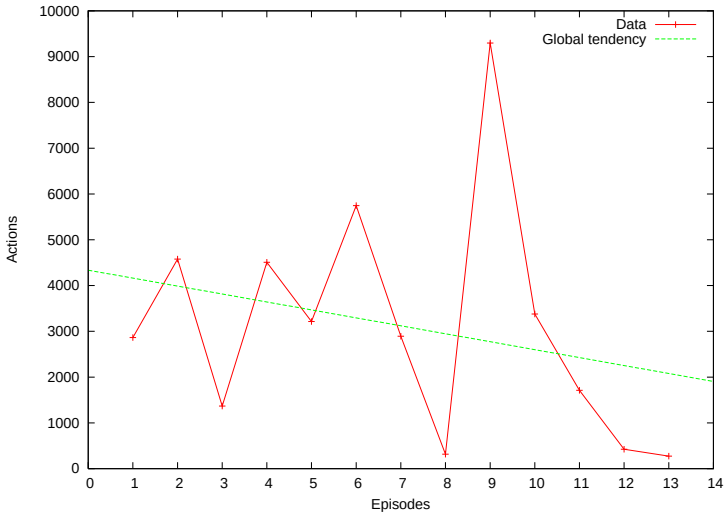

Figure 4.33: Number of actions performed before reaching the target configuration, during the learning phase (13 episodes) of a reconfiguration task policy, for experiment 7

Discussion

As can be seen in figures 4.30, 4.31, 4.32 and 4.33, the tendency of the number of actions performed during the learning episodes varies as more episodes are performed. This is because when the number of learning episodes is not enough, the approximation performed based on the existing data does not correlate to the long-term global tendency. It can also be seen that initially the slope of this curve is steeper than when more data is available (i.e. after having performed more learning episodes). This is because while more episodes are performed, the approximation is closer to the real tendency. Therefore, a fluctuation in this curve might indicate the need of performing more learning episodes.

During the observation of the robot's behaviour while using the policy obtained after 13 learning episodes, it can be seen that the robot cycles between 3 configuration, and thus cannot reach the target configuration. The obtained policy, while it always performs the same actions, is not satisfactory, since it doesn't allow to reach the target configuration. This inconvenient can probably be overcome by performing more learning episodes, in order to provide more exploratory actions that allow to determine better ways of reaching the target configuration.

Therefore, as can be observed in the presented graphics, and as could be observed during the performed simulations, the obtained behaviour is most probably due to an insufficient number of learning episodes performed.

In this occasion we decided not to continue performing learning episodes, because we considered that it was more rewarding to modify the experiment in order to use simple actions instead of complex actions. This is so because the use of simple actions reduces the size of the state-action space, allowing to find an optimal policy with a higher probability.

4.2.7 Experiment 8: From circular to linear structure, with docking/undocking, using simple actions

As mentioned previously, this experiment is basically a simplification of the previous experiment, modifying the definition of action in order to use simple actions. We pretend to evaluate the fact that not having reached an optimal policy in the previous experiment is because the number of learning episodes were not enough for the size of the state-action space. Therefore, by simplifying the actions, and thus reducing the size of the state-action space, we should be able to find an optimal policy using a relatively small amount of learning episodes.

Results

In this case, we performed 7 episodes, constituting a total of 526 simulations. The results obtained from these can be observed in figures 4.34 - 4.37.

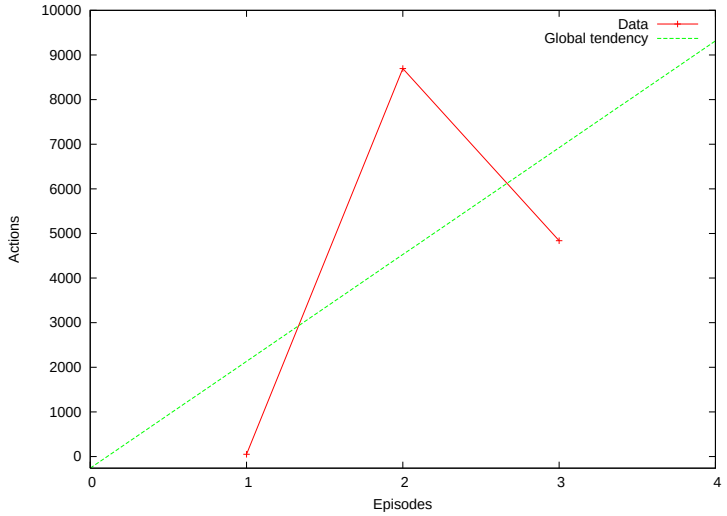

Figure 4.34: Number of actions performed before reaching the target configuration, during the learning phase (3 episodes) of a reconfiguration task policy, for experiment 8

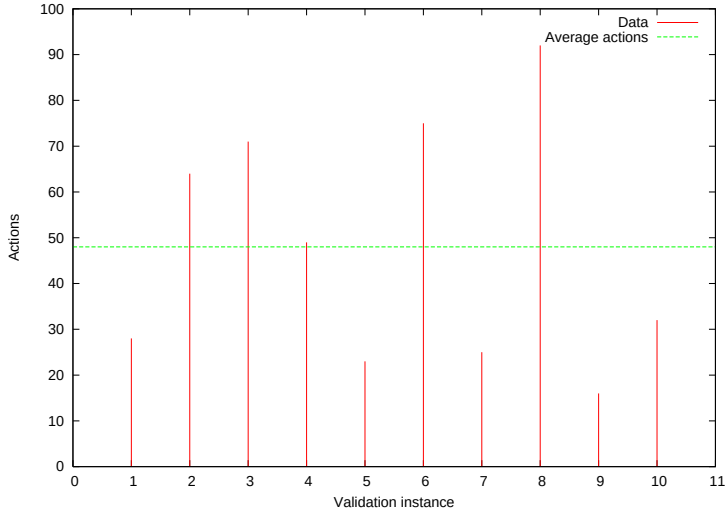

Figure 4.35: Number of actions performed before reaching the target configuration, using the learned policy after 3 episodes, for experiment 8

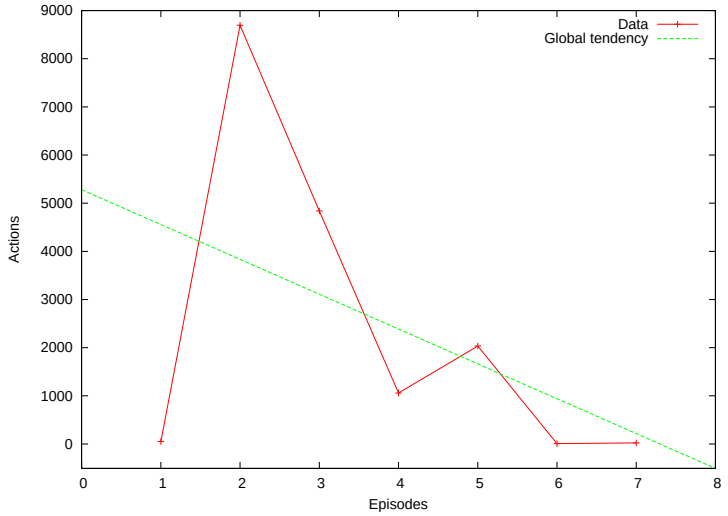

Figure 4.36: Number of actions performed before reaching the target configuration, during the learning phase (7 episodes) of a reconfiguration task policy, for experiment 8

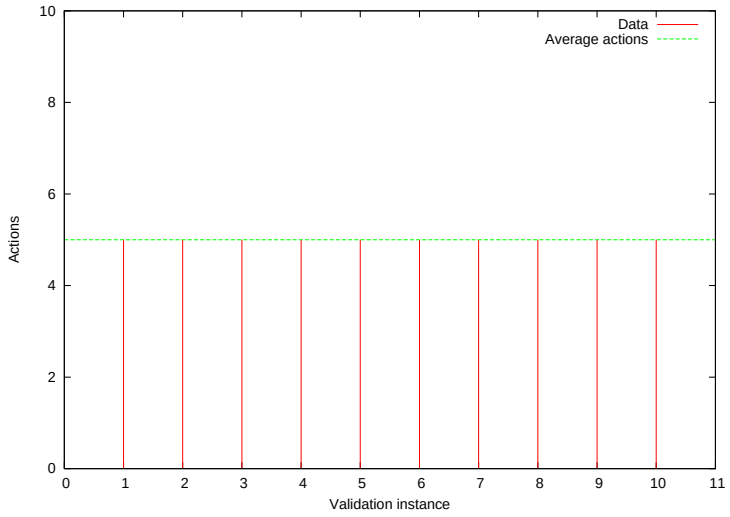

Figure 4.37: Number of actions performed before reaching the target configuration, using the learned policy after 7 episodes, for experiment 8

Discussion

As can be observed in figure 4.37, unlike the previous experiment, in this case we reached a stable policy. Even though less episodes were performed, the number of simulations was higher. The reason for this is that since having less available actions (in this experiment we used simple actions instead of complex actions), the probability of performing undocking actions (which can make the simulation to be aborted) is higher. By having many simulations be aborted, it is necessary to perform more simulations per episode, which affects the number of times the different states get visited, thus improving the robot's learning process.

4.2.8 Conclusions

In the light of the results obtained from the performed experiments, it can be seen that learning a reconfiguration task in an M-TRAN robot is feasible. Certain aspects, like the definition of an episode, the reward function, states and actions, naturally depend strongly on the problem at hand, and must therefore be refined as problems occur with the used definitions, but in the end it is possible to successfully learn reconfiguration tasks.

We can then conclude that the M-TRAN robots possesses the necessary abilities to successfully perform learning tasks in the field of reconfiguration (at least those tasks that are simple enough).

Chapter 5

Proof of concept: policy combination

After performing a diverse set of experiments that each verified the hypothesis that an M-TRAN Self-Reconfigurable Modular Robot possess the capacity of learning behaviours for solving reconfiguration and locomotion tasks by using Reinforcement Learning, we tried to push the limits a little bit and perform a proof of concept to show the real utility of this method.

A more complex experiment was defined that involves all the analyzed capacities. An obstacle was to be put in the environment that couldn't be overcome unless the robot could change it's configuration.

In this case it was decided that the way of overcoming the obstacle was by sliding below it, in a linear configuration for performing caterpillar-like locomotion. This obstacle could be representing a hole in a wall that could give access to another room in a real world scenario. This type of scenarios would be very plausible in situations like collapsed building exploration, for example.

The suggested experiment starts with the robot at one end of the experimentation environment, and configured in a wheel-like configuration. The robot has to advance until it detects the obstacle. Once detected, it should reconfigure into a linear configuration that allow it to slide under the obstacle. Once on the other side, it must again reconfigure into a wheel-like configuration in order to continue advancing.

This problem, as was just described, can be divided into several subproblems, each of which requires the robot to learn a different behaviour in order to solve them. By combining the individual behaviours, it's possible to solve the global problem, and overcome the obstacle.

The initially identified subproblems are:

1. Perform locomotion in a wheel-like configuration
2. Reconfigure from a wheel-like configuration into a linear configuration
3. Perform locomotion in a linear configuration
4. Reconfigure from a linear configuration into a wheel-like configuration

It was decided that the robot used for this experiment should be formed by 5 modules. This number is enough in order for the problem to present a greater level of complexity than the problems solved previously, in chapter 4.

For each identified subproblem a policy was obtained that would allow the robot to achieve the goal at hand (being advancing or reconfiguring itself).

Once these policies were obtained, a way had to be defined for these policies to be combined in order to attain a global and more complex behaviour that would reflect the behaviours associated with each policy, according the the different situations encountered.

When combining these policies, a problem was detected: while each policy could solve each individual problem for which they were generated, they would only be appropriate under the exact conditions under which they were acquired. That is, a reconfiguration policy could only reach the target configuration if it was evaluated starting from the same initial state as used during the learning of that policy.

Since it wasn't possible to initially determine the state in which the robot would be when one policy should stop being used and another should start, it was necessary to add new policies that would allow the robot to adapt from each of the possible final states for one policy into the initial state required by the next policy to be executed. New policies were required for:

a. Adapting the final states reached when using policy 1 into the initial state required by policy 2.
b. Adapting the final states reached when using policy 3 into the initial state required by policy 4.

Once these new policies were obtained, it was possible to combine all policies in order for the robot to present a behaviour that would allow it to overcome the obstacle and thus solve the more complex problem.

5.1 Policy combination

When combining the different policies learned independently, so to solve the global problem, it is necessary to determine which policy must be used at each moment. For this, proximity sensors were implemented that allowed the robot to detect obstacles. The robot used for this experiment would then have one proximity sensor in each module, besides all the sensors described in chapter 4.

These proximity sensors would be valued according to the following equation:

$$v = \begin{cases} 1 & \text{if } distance[i] < 0.5 * length[i] + range_s \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

where

$$\begin{aligned} distance &= \text{distance vector between the sensor and the obstacle} \\ i &= \text{dimension in which the distance is maximal} \\ length &= \text{vector of the obstacle's sides lengths} \\ range_s &= \text{sensor's maximum sensing range} \end{aligned}$$

By means of these new sensors, it's possible to determine if the robot has an obstacle ahead, above or behind itself. To determine this, the values of the sensors of the module closest to the emitting source and of the module farthest away from the emitting source are taken. The former allows to determine if an obstacle is ahead of the robot, while the latter indicates if the obstacle is behind the robot. If both sensors are activated (their value is 1), the obstacle lies above the robot.

A set of rules was defined according to the values of these sensors, that allow to determine the policy that must be used at each moment. Then, after each action is executed, these rules are evaluated and it can be determined if it's necessary to update the current policy (i.e. choosing a different policy than the one being used). In the case of the reconfiguration policies, the policy change takes place when the new configuration is reached (i.e. when the final state for those policies is reached).

As can be observed in tables 5.1 and 5.2, the policy is determined by the proximity sensor values, in the case of locomotion policies (the only policies that produce changes are policies 1 and 3, which are locomotion policies), while in the case of reconfiguration policies the second table shows the policy to be used once the target configuration has been reached (in this case, policies 1 and 3 are the ones that stay unmodified).

Current policy	Sensor values		Next policy
	Front	Rear	
1	0	0	1
1	0	1	1
1	1	0	a
1	1	1	a
a	0	0	a
a	0	1	a
a	1	0	a
a	1	1	a
2	0	0	2
2	0	1	2
2	1	0	2
2	1	1	2
3	0	0	b
3	0	1	3
3	1	0	3
3	1	1	3
b	0	0	b
b	0	1	b
b	1	0	b
b	1	1	b
4	0	0	4
4	0	1	4
4	1	0	4
4	1	1	4

Table 5.1: Rule set that define the policy to be used, according to the values of the proximity sensors

Current policy	Next policy
1	1
a	2
2	3
3	3
b	4
4	1

Table 5.2: Rule set that defines the policy to be used, after reaching the stopping condition

5.2 Results

Next we present the results of the experiments regarding obtaining the policies used in this context.

5.2.1 Policy 1: perform locomotion in a wheel-like configuration

This policy was the only one that was not obtained through learning. Basically, the amount of possible states was so overwhelming, even after introducing previous knowledge, that it would have taken too much time to learn the policy. Without a doubt, the treatment of the state space for Reinforcement Learning in this kind of robots constitutes a future and necessary line of research. Next we present an estimate for the size of the state-action space, in order to justify this decision.

The actions a module can perform can be divided into

1. Activate a motor: 8 possible actions
2. Dock with another module: 3 possible actions (for each other module)
3. Undock from another module: 3 possible actions

So, if the robot is built from 5 modules, the total amount of actions is

$$(8 + 3 * 4 + 3)^5 = 23^5 = 6436343$$

Of course this is just the upper bound, since in the problem as it was defined, it isn't necessary to dock nor undock with another module. Therefore, thanks to previous knowledge about the given problem, it's possible to eliminate those actions, and so this number gets reduced to

$$8^5 = 32768$$

In this context, the state of a module is composed by

- 2 motors, with 3 values for each motor
- 2 sensors, with 2 values for each sensor

Therefore, the number of states a module can be in is

$$2 * 2 * 3 * 3 = 36$$

and thus, the total amount of possible states for the robot is

$$36^5 = 60466176$$

We have to remember that we used previous knowledge to reduce the size of the state space (since couplings between modules are not taken into account). It's also true that this last value is an upper bound for the number of possible states, given the configuration the robot is in. Since the robot finds itself most of the time in the same relative position to the emitting source, a more realistic bound would be

$$(2 * 3 * 3)^5 = 1889568$$

By using these calculated values, we can estimate the approximate size for the state-action space to be

$$1889568 * 32768 = 6.19 * 10^{10}$$

possible values.

Even assuming one action can be performed per second, in order to fully explore the state-action space it would require

$$\begin{aligned} 6.19 * 10^{10} \text{ seconds} &\approx 1031956070 \text{ minutes} \\ &\approx 17199268 \text{ hours} \\ &\approx 716637 \text{ days} \\ &\approx 23888 \text{ months} \\ &\approx 1990 \text{ years} \end{aligned}$$

Clearly, the problem is complex. Since in order to guarantee that the learning algorithm converges to the optimal policy it's necessary that all states are visited continuously, this problem can not be treated directly as it is. In order to be able to learn a policy that solves this problem it will be necessary to further restrict it, so to further reduce the size of the state-action space, or defining a more convenient representation for states and actions. As the goal of this experiment wasn't to find the optimal policy for the locomotion task in a 5-module robot configured in a wheel-like configuration (but to use this policy for solving a more complex problem), we decided to generate the policy manually. Even though this manually generated policy might not be optimal, it correctly solves the problem.

5.2.2 Policy a: adapting the final states reached when using policy 1 into the initial state required by policy 2

To develop this policy, an iterative approach was tried. To simplify the learning process, we decided to guide the policy's evolution. So, several learning episodes were performed that consisted in learning to revert the actions performed by the previous policy. Let the initial state required by policy 2 be s_0, and let the final states reached when using policy 1 be $s_1, s_2, \ldots$. We started learning a policy for going from state s_1 to state s_0. Once this was learned, the goal was modified so that the learned policy would be extended with the knowledge of how to go from state s_2 to s_1. The same was done for each of the possible states (s_3 to s_2, s_4 to s_3, ...). As a result, we obtained a policy that was able to reach the state s_0 from any of the final states s_i.

In this way, the complexity could be kept bounded and a policy could be found that would solve the problem in an adequate way.

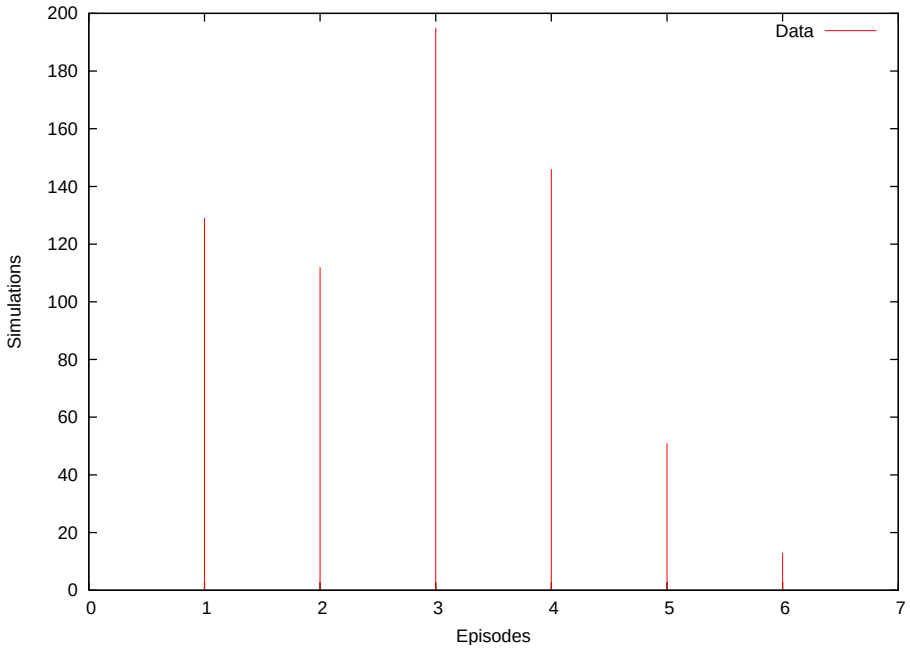

Figure 5.1: Number of simulations performed per learning episode of policy *a*.

In this case, since each learning episode constituted a different problem, and therefore an independent problem, it doesn't make sense the calculate a "global tendency" as in the other experiments. Because of this, figures 5.1

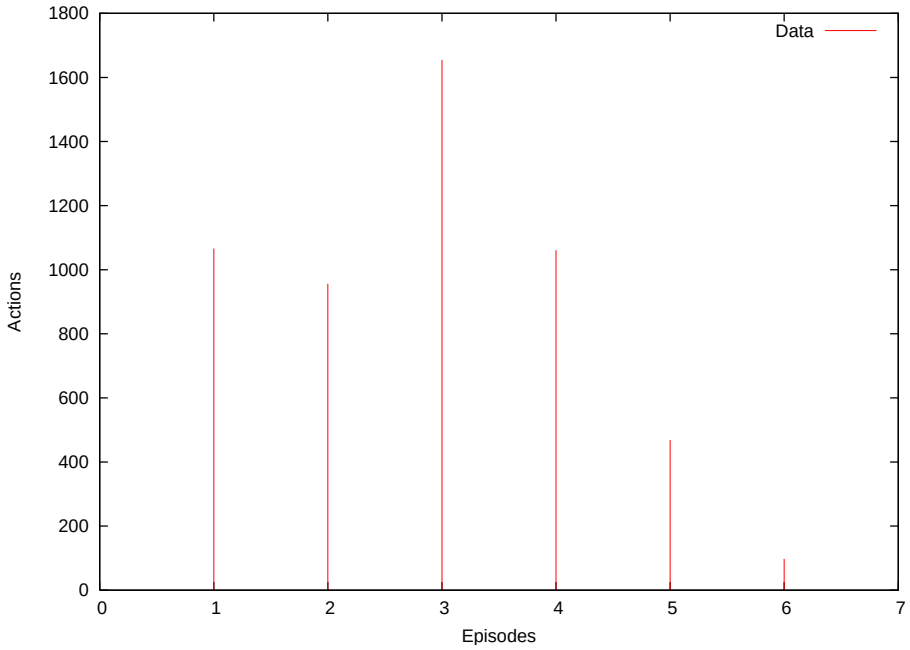

Figure 5.2: Number of actions performed during each learning episode of policy a.

and 5.2 only show the number of simulations and actions required by each learning episode.

5.2.3 Policy 2: reconfigure from a wheel-like configuration into a linear configuration

In this case, 7 learning episodes were performed, adding up a total of 332 simulations, during which 3038 actions were executed.

The treated problem was simple enough so as not having to reduce the state-action space.

As can be seen in figures 5.3 - 5.6, at the end of the learning episodes, a policy was found that efficiently solved the reconfiguration task.

5.2.4 Policy 3: perform locomotion in a linear configuration

For the next policy, 13 learning episodes were performed, constituted by 16 simulations, that encompassed a total of 2243 actions. In this case, due to the big state-action space size, it was necessary to reduce the latter, simplifying

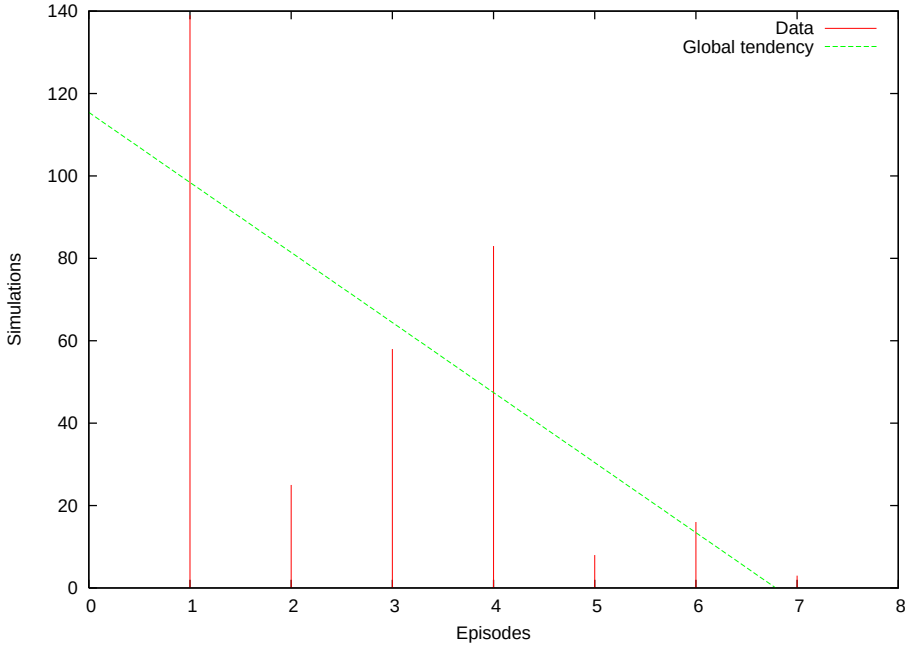

Figure 5.3: Number of simulations required for each learning episode of policy *2*.

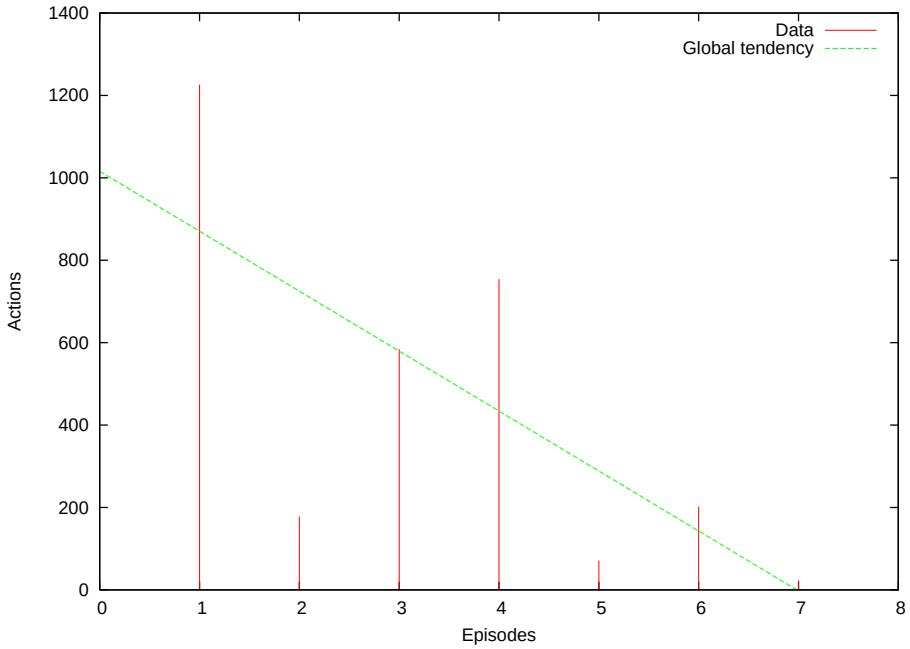

Figure 5.4: Number of actions performed during each learning episode of policy *2*.

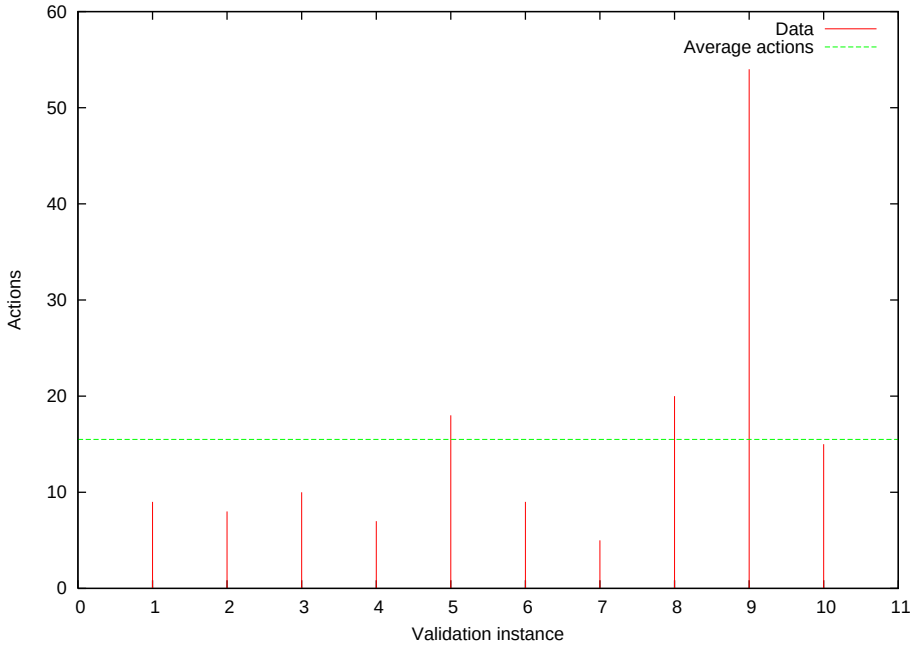

Figure 5.5: Number of actions performed by policy *2* before reaching the target configuration, during the validation phase, after 3 learning episodes.

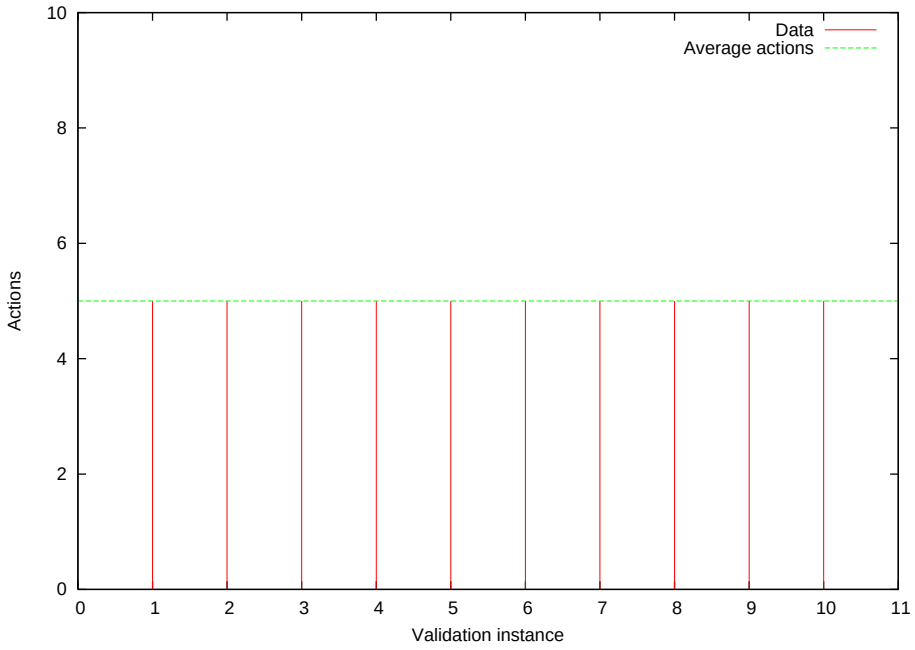

Figure 5.6: Number of actions performed by policy *2* before reaching the target configuration, during the validation phase, after 7 learning episodes.

the problem. The available actions were limited so that it was only possible to activate 2 motors: the motor of the passive block nearest to the emitting source, and the motor of the active block farthest away from it. These blocks form both ends of the robot.

This decision was based on the fact that a policy could be manually generated that would solve the locomotion task by just using those motors. So, we proceeded to learn a policy that showed a similar behaviour to the manually generated policy. In this way a policy was found through learning that could achieve locomotion in a linear configuration. It's expected that the generated policy is not optimal regarding the robot's locomotion capabilities (i.e. if the actions wouldn't have been limited), but it resulted good enough for this experiment.

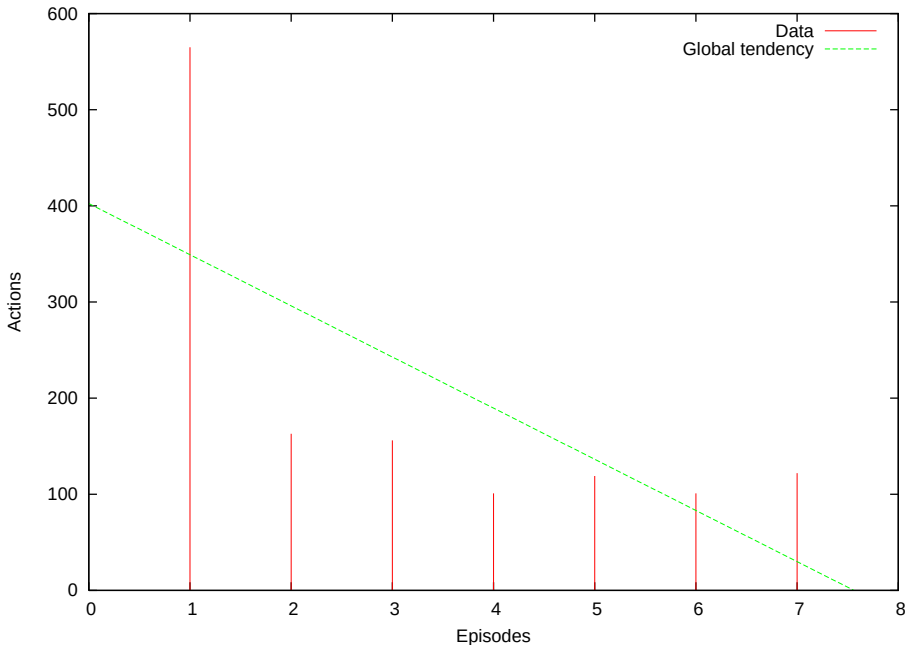

Figure 5.7: Number of actions performed before reaching the goal, during the learning phase (7 episodes) of policy *3*.

5.2.5 Policy b: adapting the final states reached when using policy 3 to the initial state required by policy 4

To determine the correct policy for the problem of adapting policy 3 to policy 4, 36 learning episodes had to be performed. In this occasion only one

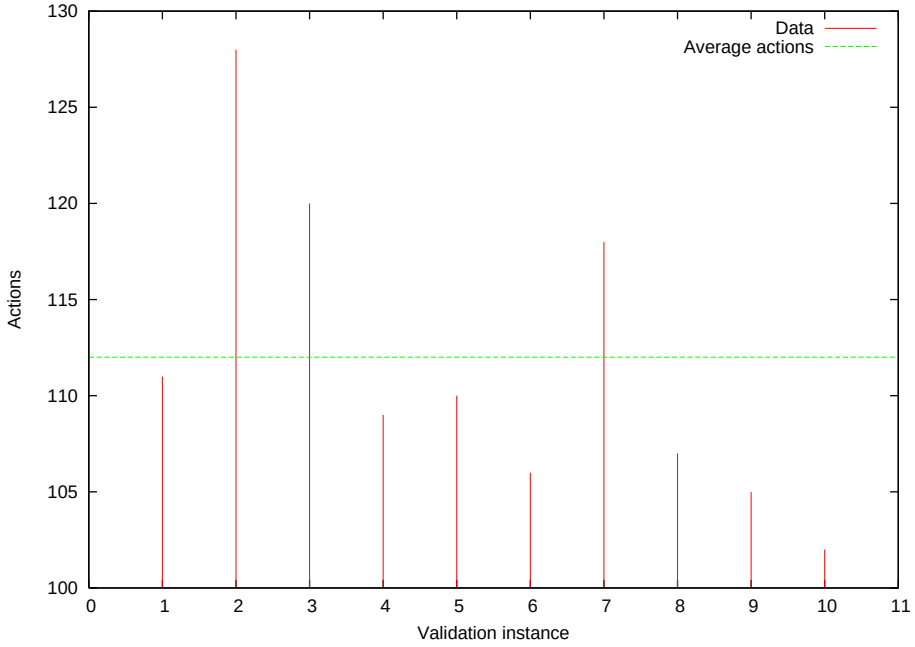

Figure 5.8: Number of actions performed before reaching the goal, during the validation phase for policy *3*, after 7 learning episodes.

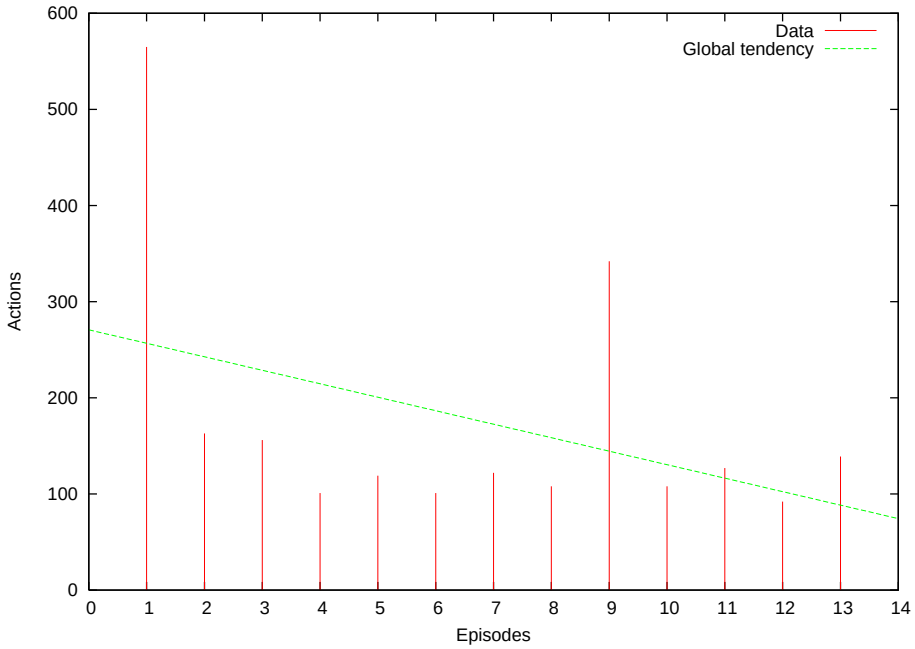

Figure 5.9: Number of actions performed before reaching the goal, during the learning phase (13 episodes) of policy *3*.

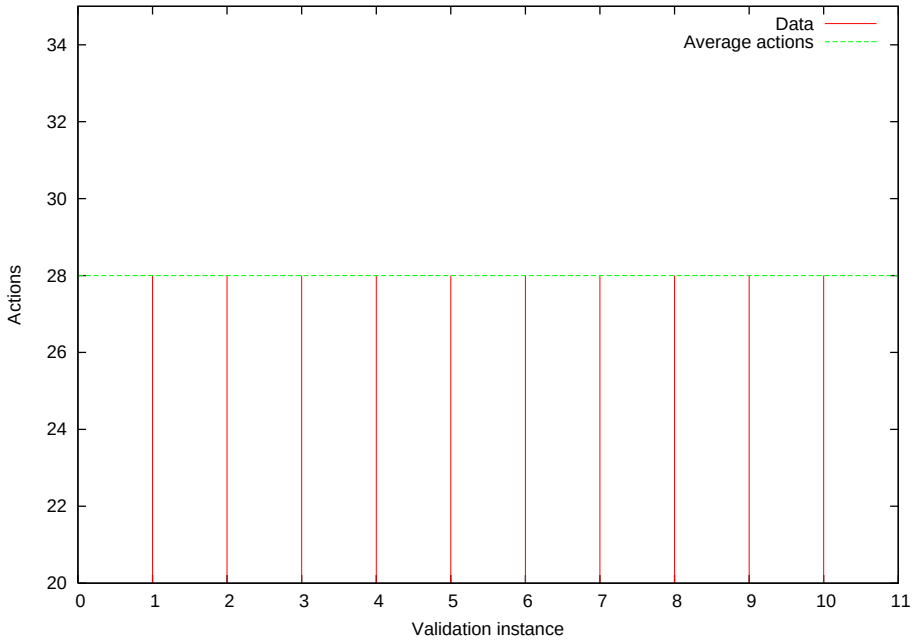

Figure 5.10: Number of actions performed before reaching the goal, during the validation phase for policy *3*, after 13 learning episodes.

simulation was needed per episode, and a total of 170 actions were performed.

This time we proceeded in a similar way as in the case of policy a. The learned policy was constructed incrementally, but instead of guiding the solution, we proceeded to extend the policy by directly evaluating the reconfiguration between each possible final state of policy 3 and the initial state of policy 4. This distinction was made because in this case the problem was simpler than in the previous case.

This last assertion can be appreciated in the results obtained during the learning phase, where for each learning episode only 6 actions were performed on average.

5.2.6 Policy 4: reconfigure from a linear configuration into a wheel-like configuration

Finally, for the last policy, 3 learning episodes were performed, adding up a total of 126 simulations and performing a total of 3207 actions.

In this case, the policy was rapidly obtained (only a few actions were necessary during each learning episode) and it turned out to be efficient in solving the reconfiguration task, which can be observed in the few number of actions it requires to reach the target configuration.

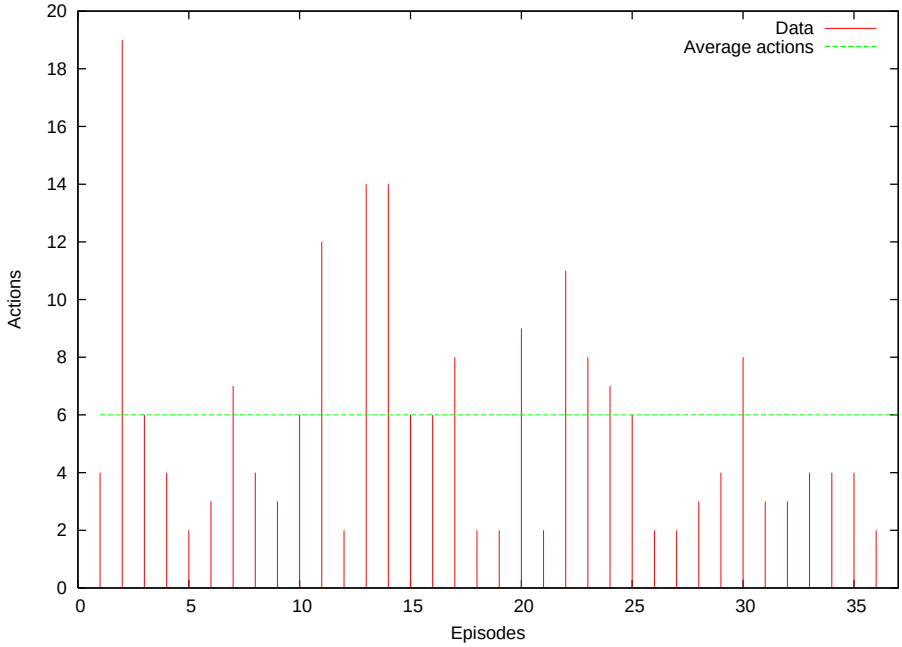

Figure 5.11: Number of actions performed during each learning episode of policy *b*.

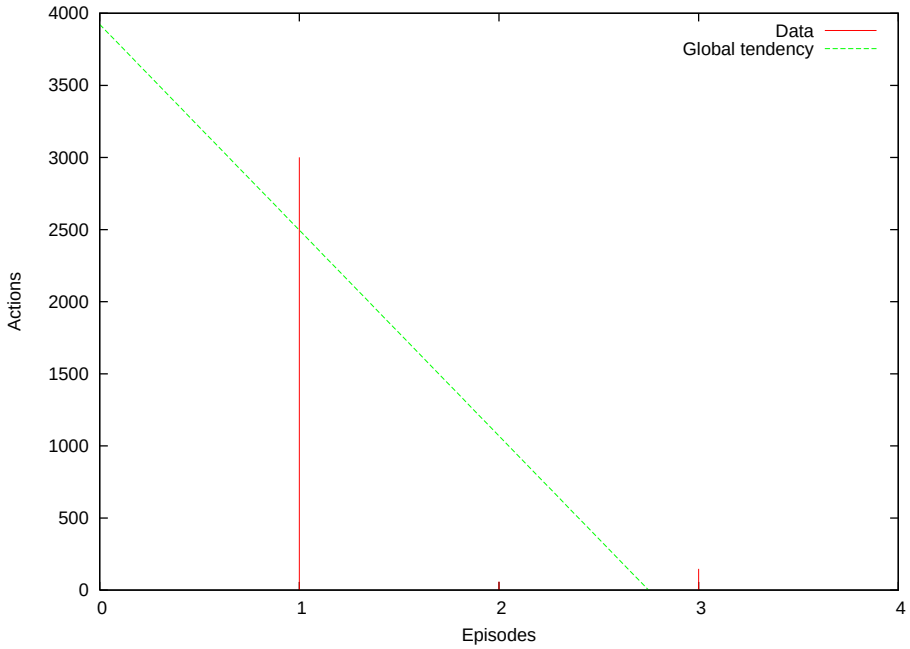

Figure 5.12: Number of actions performed before reaching the target configuration, during the learning phase (3 episodes) of policy *4*.

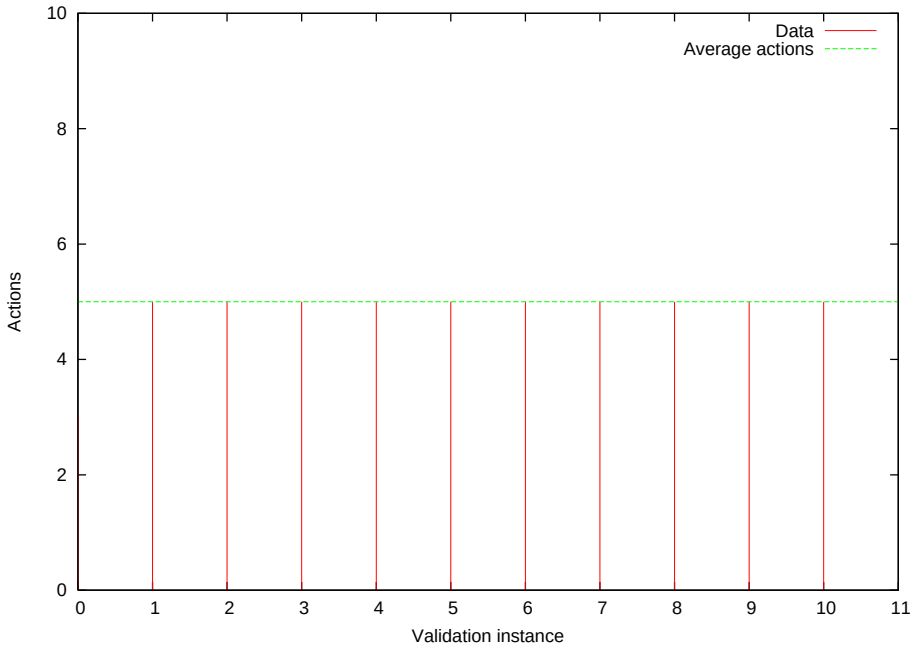

Figure 5.13: Number of actions performed before reaching the target configuration, during the validation phase of policy *4*, after 3 learning episodes.

5.3 Conclusions

In spite of the inconvenients during the learning of some policies, it could be seen that it's possible to combine simple policies learned independently, and obtain as a result a more complex behaviour.

We detected an inconvenient due to the fact that the initially developed policies strongly depended on the initial state. This inconvenient can be easily avoided if all possible initial states are taken into account when first learning the policy (as it was later done with the policies it was necessary to add in order to fix this error), so this does not really represent a problem.

Even though not all policies were obtained through learning, this doesn't either represent a problem since

1. it is possible to learn those policies, by correctly restricting the size of the state-action space. In this occasion it wasn't considered relevant to spend the greater amount of time it takes to learn those more complex policies, since the way they are obtained does not alter the result of the experiment.

2. the mechanism used for combining the policies does not takes into account the way in which the policies were acquired, so it's possible

to combine policies learned, evolved, manually generated or acquired in any other way.

This last fact can be considered as a positive characteristic of the method, because in certain cases it's not vital to acquire a policy through learning, and it can even be better to obtain the policy in some other way; for example, it's possible to obtain an optimal policy manually, when the behaviours involved are sufficiently simple. In this cases, it can be better to use those manually generated policies, and dedicate the saved time to learn a policy for a more complex behaviour.

As a final conclusion, it is clear that it's perfectly possible to use policies associated to simple behaviours, and by combining them produce as a result a policy that describes a more complex behaviour. This is good, since directly obtaining a policy for the global behaviour can sometimes be extremely difficult (both when trying to define it manually or if a computational approach is used, like learning or evolution).

Chapter 6

Conclusions

In the current work we presented an overview of the state of the art in the field of Self-Reconfigurable Modular Robots, and developed a simulator with which experiments were performed relative to the use of the Reinforcement Learning methodology applied to the problem of obtaining behaviours that would allow to solve locomotion and reconfiguration tasks with Self-Reconfigurable Modular Robots. For each type of behaviour to be learned the most adequate representation for the state-action space had to be studied and the conditions under which Reinforcement Learning was possible had to be defined. The performed analysis constituted a first approach to the problem that was presented in this work, from which it was possible to extract some characteristics inherent to the problem at hand, and determine factors to be taken into account in order to continue working in this field. Finally, a problem was proposed that combined locomotion and reconfiguration tasks, and was resolved by means of the studied methodology.

During the analysis of the state of the art in the field of Self-Reconfigurable Modular Robots, it could be seen that effectively, no mention could be found of using Reinforcement Learning as a methodology for solving locomotion and reconfiguration tasks in this type of robots.

Through the performed experiments it could be verified that the developed simulator fulfilled the requirements to be used in experiments dealing with Reinforcement Learning applied to obtaining adequate behaviours for solving locomotion and reconfiguration tasks. It was also shown that it was possible to learn efficient policies for performing locomotion and reconfiguration tasks in Self-Reconfigurable Modular Robots (at least in the case of the M-TRAN robot).

Most of the performed experiments were successful. Those that couldn't be successfully solved, showed that the size of the state-action space was an important obstacle for obtaining acceptable results, and that a more general

treatment of this problem constitutes a future line of research.

By means of the performed experiments it was shown that it is feasible to use the methodology of Reinforcement Learning in order to learn efficient policies. It was also shown that it's possible to combine simple and independently learned policies and obtain as a result a more complex behaviour. This last fact is of great importance, since thanks to it, it's not indispensable to solve the more complex tasks directly, which can be very difficult due to the size of the state-action space associated to the more complex problems; on the contrary, it is possible to divide the complex problem into subproblems that are easier to solve, learn an adequate policy for each subproblem, and then combine those policies in order to solve the global problem.

It is clear then that the proposed methodology can be useful in this area, and that effectively it's necessary to extend the research done in this context, in order to determine better ways of allowing these robots to adapt to environmental changes and through this, make maximal use of their intrinsic capabilities.

Some possible future lines of research include:

- Study the way of reducing the size of the state-action space, in order to be able to work with bigger robots and/or with more complex problems.
- Generalize the performed experiments, in order to work with robots with more complex structures.
- Determine ways to learn how to combine policies, instead of having to manually specify how the combination is done.
- Study ways of learning to adapt to changing environmental conditions.
- Study ways of learning to adapt to different perception capabilities (adding/removing sensors).

Appendix A

Reconfigurable Modular Robot Comparison

	DOF	Homogen.	3D	Type	Self-Reconf.	Coupling
CONRO	2	✓	✓	Chain	✓	Mechanical/Bipartite
Polybot	1	✓	✓	Chain	✓	Mechanical/Bipartite
ACM	1-3	✓	✓	Chain	×	Mechanical
Metamorphic	3	✓	×	Lattice	✓	Mechanical/Bipartite
Crystalline	2	✓	✓	Lattice	✓	Mechanical/Bipartite
Fractum	0	✓	×	Lattice	✓	Magnetical/Bipartite
Micro Unit	2	✓	×	Lattice	✓	Mechanical/Bipartite
RIKEN Vertical	2	✓	×	Lattice	✓	Magnetical/Bipartite
Telecube	6	✓	✓	Lattice	✓	Magnetical/Bipartite
MEL 3D Unit	12	✓	✓	Lattice	✓	Mechanical/Bipartite
Molecule	4	✓	✓	Lattice	✓	Mechanical/Bipartite
M-TRAN (I y II)	2	✓	✓	Hybrid	✓	Magnetical/Bipartite
M-TRAN III	2	✓	✓	Hybrid	✓	Mechanical/Bipartite
I(ces)-Cubes	3	×	✓	Lattice	✓	Mechanical/Bipartite
Fracta	12	✓	✓	Lattice	✓	Mechanical/Hermaphrodite
Proteo	0	✓	✓	Lattice	✓	Magnetical/Bipartite
Miniaturized	2	✓	×	Lattice	✓	Mechanical/Bipartite
Semi-Cylindrical	2	✓	✓	Lattice	✓	Magnetical/Bipartite
TETROBOT	3-5	✓	✓	Lattice	×	Mechanical/Bipartite
CEBOT	1-3	✓	×	Lattice	×	Bipartite

Table A.1: Reconfigurable Modular Robot Comparison

Appendix B

Experimental Results

Parameters			Distance Advanced	Distance Travelled	Actions Executed	Finished Learning
alpha	epsilon	gamma				
0.1	0.5	0.5	22.55	168.65	172	✓
0.2	0.5	0.5	24.60	181.30	190	✓
0.3	0.5	0.5	23.40	180.70	181	✓
0.4	0.5	0.5	18.85	188.80	192	✓
0.5	0.5	0.5	22.20	191.40	199	✓
0.6	0.5	0.5	25.80	174.60	181	✓
0.7	0.5	0.5	23.25	172.75	178	✓
0.8	0.5	0.5	22.55	179.45	183	✓
0.9	0.5	0.5	19.40	186.10	189	✓

Table B.1: Results obtained during the learning phase for the locomotion task proposed in section 4.1, for different values of the parameter *alpha*

Parameters			Distance	Distance	Actions	Finished
alpha	epsilon	gamma	Advanced	Travelled	Executed	Learning
0.5	0.1	0.5	29.80	73.50	78	✓
0.5	0.2	0.5	29.75	104.10	105	✓
0.5	0.3	0.5	29.55	122.05	125	✓
0.5	0.4	0.5	29.65	143.05	142	✓
0.5	0.5	0.5	22.35	179.45	181	✓
0.5	0.6	0.5	14.30	194.50	201	×
0.5	0.7	0.5	12.10	196.95	200	×
0.5	0.8	0.5	4.90	188.10	201	×
0.5	0.9	0.5	5.90	176.55	201	×

Table B.2: Results obtained during the learning phase for the locomotion task proposed in section 4.1, for different values of the parameter *epsilon*

Parameters			Distance	Distance	Actions	Finished
alpha	epsilon	gamma	Advanced	Travelled	Executed	Learning
0.5	0.5	0.1	26.55	166.05	173	✓
0.5	0.5	0.2	24.55	169.10	179	✓
0.5	0.5	0.3	25.65	171.10	177	✓
0.5	0.5	0.4	22.85	183.15	187	✓
0.5	0.5	0.5	23.15	184.35	191	✓
0.5	0.5	0.6	29.35	162.75	162	✓
0.5	0.5	0.7	26.75	180.35	177	✓
0.5	0.5	0.8	23.85	168.30	170	✓
0.5	0.5	0.9	23.40	176.40	181	✓

Table B.3: Results obtained during the learning phase for the locomotion task proposed in section 4.1, for different values of the parameter *gamma*

Parameters			Distance	Distance	Actions	Performance
alpha	epsilon	gamma	Advanced	Travelled	Executed	
0.1	0.5	0.5	29.50	70.50	63	0.23247
0.2	0.5	0.5	29.50	70.50	63	0.23247
0.3	0.5	0.5	22.50	65.50	148	0.12850*
0.4	0.5	0.5	29.50	71.00	63	0.22962
0.5	0.5	0.5	29.50	71.50	63	0.22962
0.6	0.5	0.5	26.50	76.00	137	0.15608*
0.7	0.5	0.5	29.50	75.70	62	0.21981
0.8	0.5	0.5	29.50	70.50	63	0.23247
0.9	0.5	0.5	29.50	70.50	63	0.23247
0.5	0.1	0.5	29.50	63.50	62	0.26225
0.5	0.2	0.5	29.50	73.00	79	0.19539
0.5	0.3	0.5	29.50	78.00	102	0.19117*
0.5	0.4	0.5	27.00	90.00	114	0.15824*
0.5	0.5	0.5	29.50	70.50	63	0.23247
0.5	0.6	0.5	29.50	70.50	63	0.23247
0.5	0.7	0.5	29.50	62.00	56	0.35310
0.5	0.8	0.5	29.50	70.50	63	0.23246
0.5	0.9	0.5	29.50	74.50	59	0.23490
0.5	0.5	0.1	29.50	70.50	63	0.23247
0.5	0.5	0.2	29.50	70.50	63	0.23247
0.5	0.5	0.3	29.50	70.50	63	0.23247
0.5	0.5	0.4	29.50	75.50	62	0.21988
0.5	0.5	0.5	29.50	71.00	63	0.22884
0.5	0.5	0.6	29.50	57.00	98	0.30714*
0.5	0.5	0.7	29.50	75.50	62	0.22057
0.5	0.5	0.8	29.50	72.00	66	0.22132
0.5	0.5	0.9	29.50	83.50	88	0.18831*

* when using these parameters, some simulations had to be terminated due to the limit for the maximum allowed number of actions.

Table B.4: Results obtained during the evaluation of the learned policies for the locomotion task proposed in section 4.1

Appendix C

M-TRAN II and M-TRAN III Images

C.1 M-TRAN II

Figure C.1: Locomotion in an easy terrain.

Figure C.2: Reconfiguration to overcome an obstacle.

Figure C.3: Reconfiguration for climbing.

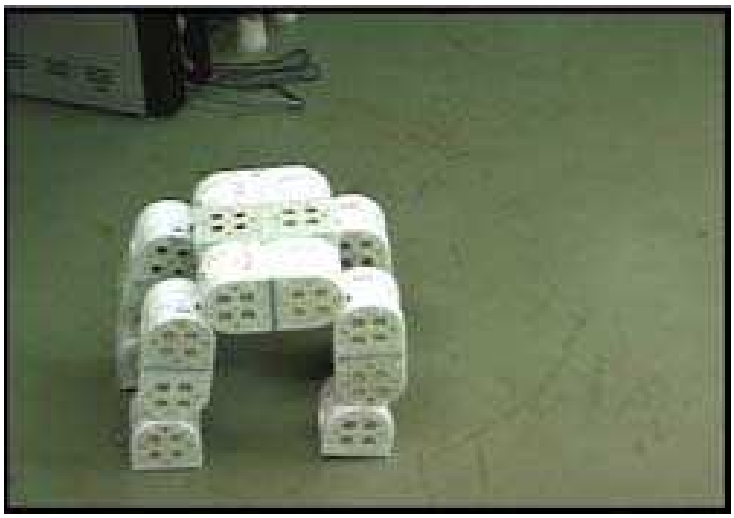

Figure C.4: Quadruped

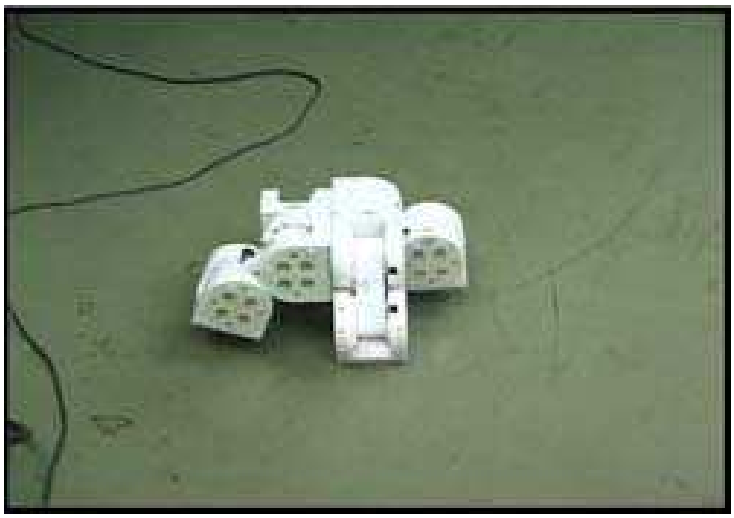

Figure C.5: Minimal version of a quadruped robot.

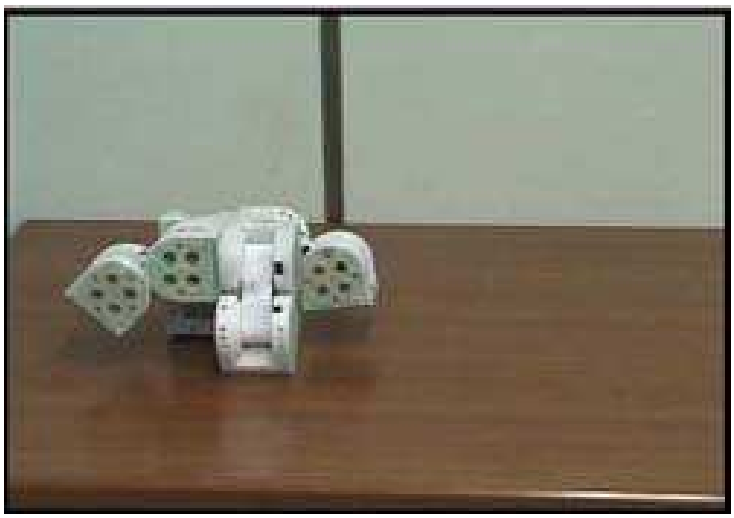

Figure C.6: Minimal version of a quadruped robot.

Figure C.7: Configuration for caterpillar-like locomotion.

Figure C.8: Configuration for caterpillar-like locomotion.

Figure C.9: Configuration for hexapod-shape locomotion.

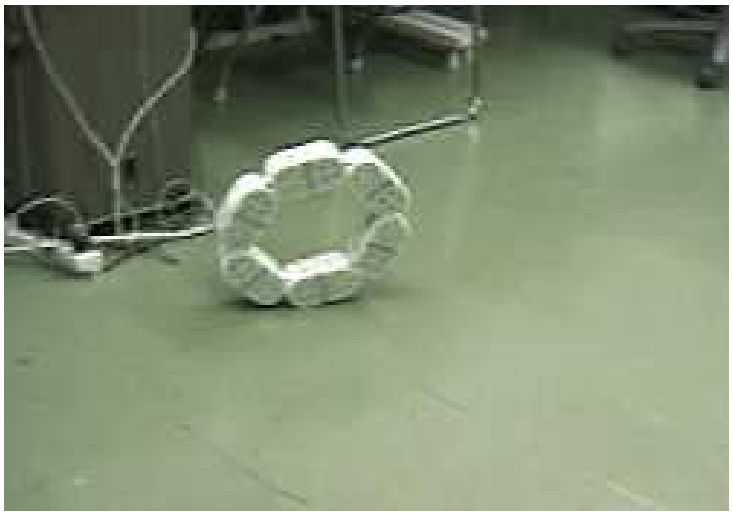

Figure C.10: Configuration for wheel-like locomotion.

Figure C.11: Due to reconfiguration it is possible to overcome obstacles.

Figure C.12: Quadruped locomotion.

Figure C.13: Configuration for snake-like locomotion.

Figure C.14: Arachnid-like configuration.

C.2 M-TRAN III

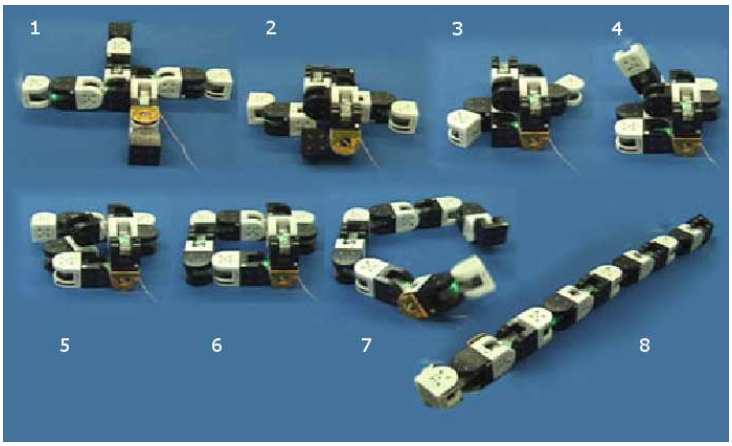

Figure C.15: Reconfiguration from quadruped to lineal (8 modules).

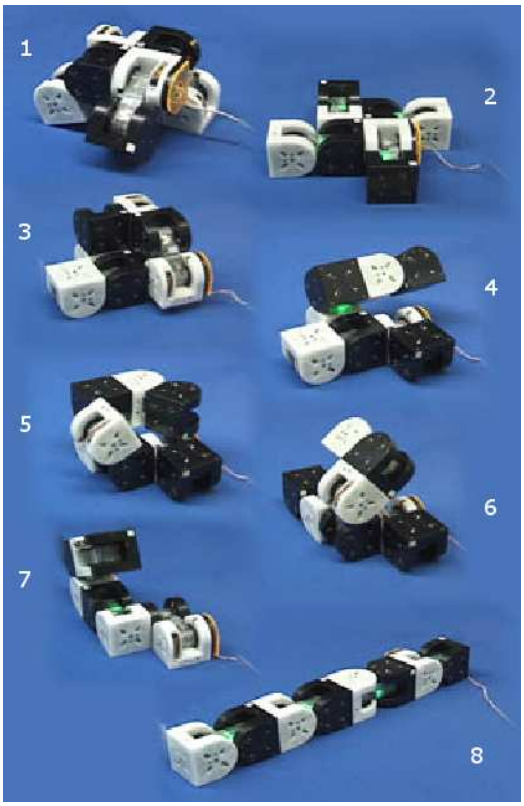

Figure C.16: Reconfiguration from quadruped to lineal (4 modules).

Bibliography

[Bax97] Baxter, J., Tridgell, A., y Weaver, L., "Knightcap: A chess program that learns by combining $td(\lambda)$ with minimax search", 1997.

[But02] Butler, Z., Kotay, K., Rus, D., y Tomita, K., "Generic decentralized control for a class of self-reconfigurable robots", *Proc. of IEEE IRCA*, págs. 809–815, 2002.

[Cas02] Castano, A., Behar, A., y Will, P., "The Conro Modules for Reconfigurable Robots", *IEEE/ASME TRANSACTIONS ON MECHATRONICS*, 7(4), 2002.

[Cri98] Crites, R. H. y Barto, A. G., "Elevator group control using multiple reinforcement learning agents", 1998.

[Dit04] Dittrich, E., "Modular Robot Unit - Characterisation, Design and Realisation", 2004.

[Jan01] Jantapremjit, P. y Austin, D., "Design of a Modular Self-Reconfigurable Robot", *Proceedings of the 2001 Australian Conference on Robots and Automation*, págs. 38–43, 2001.

[Jor04] Jorgensen, M. W., Ostergaard, E. H., y Lund, H. H., "Modular ATRON: Modules for a self-reconfigurable robot", *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004.

[Kam03] Kamimura, A., Kurokawa, H., Yoshida, E., Tomita, K., Murata, S., y Kokaji, S., "Automatic Locomotion Pattern Generation for Modular Robots", *Proceedings of the 2003 IEEE International Conference on Robotics and Automation*, 2003.

[Kot98] Kotay, K., Rus, D., Vona, M., y McGray, C., "The Self-reconfiguring Robotic Molecule: Design and Control Algorithms", *International Conference on Robotics and Automation*, 1998.

[Kum98] Kumar, G. P. y Venkataram, P., "Network performance tuning using reinforcement learning", 1998.

[Kur02] Kurokawa, H., "Self-reconfigurable Modular Robot (M-TRAN) and its Motion Design", *Proc. ICARCV 2002*, págs. 51–56, 2002.

[Mur94] Murata, S., Kurokawa, H., y Kokaji, S., "Self-Assembling Machine", *Proc. IEEE Int. Conf. on Robotics and Automation*, págs. 441–448, 1994.

[Mur02] Murata, S., Yoshida, E., Kamimura, A., Kurokawa, H., Tomita, K., y Kokaji, S., "M-TRAN: Self-Reconfigurable Modular Robotic System", *IEEE/ASME Transactions on Mechatronics*, 7(4):431–441, 2002.

[Nil02] Nilsson, M., "Connectors for Self-Reconfiguring Robots", *IEEE/ASME Transactions on Mechatronics*, 7(4):473–474, 2002.

[Nol98] Nolfi, S. y Floreano, F., "How co-evolution can enhance the adaptive power of artificial evolution: implications for evolutionary robotics", *Proceedings of EvoRobot'98*, págs. 2–8, 1998.

[ODE] "Open Dynamics Engine", `http://www.ode.org/`.

[Pat04] Patterson, S. A., Knowles, K. A. J., y Bishop, B. E., "Towards Magnetically-Coupled Reconfigurable Modular Robots", *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, 2004.

[Rou00] Roufas, K., Zhang, Y., Duff, D., y Yim, M., "Six Degree of Freedom Sensing for Docking Using IR LED Emitters and Receivers", 2000.

[Rus00] Rus, D. y Vona, M., "A basis for self-reconfigurable robots using crystal modules", *Proc. IEEE IROS*, págs. 2194–2202, 2000.

[She] Shen, W.-M., Salemi, B., y Will, P., "Hormones for Self-Reconfigurable Robots", URL `http://www.isi.edu/conro/`.

[Sto02] Stoy, K., Shen, W.-M., y Will, P. M., "Using Role-Based Control to Produce Locomotion in Chain-Type Self-Reconfigurable Robots", *IEEE/ASME Transactions on Mechatronics*, 7(4):410–417, 2002.

[Sto03] Stoy, K., Shen, W.-M., y Will, P. M., "Implementing Configuration Dependent Gaits in a Self-Reconfigurable Robot", *Proceedings of the 2003 IEEE International Conference on Robotics and Automation*, págs. 3828–3833, 2003.

[Sto04] Stoy, K., "Emergent Control of Self-Reconfigurable robots", 2004.

[Suh02] Suh, J. W., Homans, S. B., y Yim, M., "Telecubes: Mechanical design of a module for self-reconfigurable robotics", *Proceedings on the IEEE International Conference on Robotics and Automation*, págs. 4095–4101, 2002.

[Sut98] Sutton, R. y Barto, A., *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998.

[Tes95] Tesauro, G., "Temporal difference learning and td-gammon", *Communications of the ACM*, 38:58–68, 1995.

[Üns99] Ünsal, C., Kiliççöte, H., y Khosla, P. K., "I(CES)-cubes: a modular self-reconfigurable bipartite robotic system", *Proceedings of SPIE*, 3839:258–269, 1999.

[Wat89] Watkins, C., *Learning from Delayed Rewards*, Tesis Doctoral, King's College, Oxford, 1989.

[Yim00] Yim, M., Duff, D., y Roufas, K., "PolyBot: a Modular Reconfigurable Robot", *Proc. of the IEEE Int. Conf. on Robotics and Automation*, págs. 1734–1741, 2000.

[Yim02] Yim, M., Zhang, Y., y Duff, D., "Modular Robots", *IEEE Spectrum*, págs. 30–34, 2002.

[Yos01a] Yoshida, E., Murata, S., Kamimura, A., Tomita, K., Kurokawa, H., y Kokaji, S., "A motion planning method for a self-reconfigurable modular robot", *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, págs. 590–597, 2001.

[Yos01b] Yoshida, E., Murata, S., Kamimura, A., Tomita, K., Kurokawa, H., y Kokaji, S., "Reconfiguration Planning for a Self-Assembling Modular Robot", *Proceedings of the 4th IEEE International Symposium on Assembly and Task Planning*, 2001.