THESIS

Presented by

Frédéric GRUAU

in front of l'ECOLE NORMALE SUPERIEURE DE LYON
l'UNIVERSITE CLAUDE BERNARD-LYON I

for the obtention
of a DIPLOME DE DOCTORAT
Specialty: Computer Science

NEURAL NETWORK SYNTHESIS USING CELLULAR ENCODING AND THE GENETIC ALGORITHM.

Date of defense:	January 4th, 1994	
Composition of the Jury:	Président:	M. Maurice Nivat
	Reviewers:	M. Jean-Arcady Meyer M. Jacques Demongeot
	Examinators:	M. Michel Cosnard M. Jacques Mazoyer M. Pierre Peretto M. Darell Whitley

Phd Thesis prepared at Centre d'étude nucléaire de Grenoble
within the Département de Recherche Fondamentale matière condensée
and at l'Ecole Normale Supérieure de Lyon,
in the Laboratoire de l'Informatique du Parallélisme (LIP-IMAG),
Unité de Recherche Associée au CNRS

Acknowledgments

I am very grateful to Michel Cosnard who is my thesis advisor. He gave me his trust from before the beginning until after the end of my thesis. On can summerize the efficiency of Michel Cosnard in one equation: 10 minutes of Michel Cosnard = one week of work and money. Michel Cosnard gave me the passion for scientif research, communication and for visiting other laboratories. Michel Cosnard was the first in France to be interested in Genetic Neural Networks. I believe that this subject is going to become the trunc of the connectionism. Although Michel Cosnard is at the origine of many of the ideas in this thesis, he has never put his name on any of my publications. I thank Pierre Peretto, which was responsible for me, in front of the CEA. He helped me a lot. He has corrected my first publication. He made me discover a new world: statistical physics. He always has encouraged me, and has given me freedom. Thanks to him, I could work in ideal conditions. I thank Mirta Gordon with whom I had many fruitfull talks. I had a great cooperation with Darell Whitley, at the Colorado state university in Fort Collins, about the combination of learning and GAs. Darell Whitley was very open to discussions, and he helped me a lot to present cellular encoding. I acknowlege that he has written most of chapter 6 of this thesis. I thank Heinz Müehlenbein who wellcame me in his laboratory for one month. Heinz Müehlenbein gave me good advices. I thank John Koza who generously told me about the results and the ideas of its book "GPII", two years before its publication. I like such things.

Abstract

Artificial neural networks used to be considered only as a machine that learns using small modifications of internal parameters. Now this is changing. Such learning method do not allow to generate big neural networks for solving real world problems. This thesis defends the following three points:

- The key word to go out of that dead-end is ”modularity”.
- The tool that can generate modular neural networks is cellular encoding.
- The optimization algorithm adapted to the search of cellular codes is the genetic algorithm.

The first point is now a common idea. A modular neural network means a neural network that is made of several sub-networks, arranged in a hierarchical way. For example, the same sub-network can be repeated. This thesis encompasses two parts. The first part demonstrates the second point. Cellular encoding is presented as a machine language for neural networks, with a theoretical basis (it is a parallel graph grammar that checks a number of properties) and a compiler of high level language. The second part of the thesis shows the third point. Application of genetic algorithm to the synthesis of neural networks using cellular encoding is a new technology. This technology can solve problems that were still unsolved with neural networks. It can automatically and dynamically decompose a problem into a hierarchy of sub-problems, and generate a neural network solution to the problem. The structure of this network is a hierarchy of sub-networks that reflects the structure of the problem. The technology allows to experience new scientific domains like the interaction between learning and evolution, or the set up of learning algorithms that suit the GA.

Contents

1 Introduction to Genetic Neural Networks **1**

1.1 Genetic Algorithms 1
1.2 Genetic Programming 1
1.3 Neural networks 2
1.4 Genetic Neural Networks 2
1.5 Overview 4

2 Presentation of Cellular Encoding **7**

2.1 Basic Cellular Encoding 7
2.2 Coding of recursivity 9
2.3 Comparison with Kitano 's scheme 11

3 Properties of Cellular Encoding **15**

3.1 Introduction 15
3.2 Architecture encoding 15
3.3 Neural Network Encoding 21
3.4 Simulation of a Turing machine 22
3.5 Conclusion 30

4 A neural Compiler **31**

4.1 Introduction 31
4.2 The stage of the compilation 31
4.3 The kind of neurons which are used 33
4.3.1 Sigmoid 33
4.3.2 Dynamic 33
4.4 Microcoding 34
4.5 New program symbols 35
4.6 Principles of the neural compiler 37
4.6.1 Structure of the compiled neural network 37
4.6.2 Compilation of a Pascal Instruction 37
4.6.3 Compilation of the Pascal program 39
4.7 The macro program symbols 41
4.7.1 Kernel of the Pascal 43
4.7.2 Procedures and functions 45
4.7.3 The arrays 46
4.7.4 The enhanced Pascal 47
4.8 Results of the compilation 48

4.8.1 Compilation of a standard Pascal program 48
4.8.2 How to use the `CALLGEN` instruction. 50
4.8.3 How to use the #IF instruction 51
4.9 Application of the compiler 52
4.9.1 Neural network design 52
4.9.2 Tool for the design of hybrid systems 53
4.9.3 Automatic parallelization 53

5 Genetic synthesis of Neural Networks without learning **57**
5.1 Genetic Programming 57
5.2 The fitness 59
5.3 Simulation with the parity 61
5.4 Simulation with the symmetry 63
5.5 Conclusion 68

6 Interaction between learning and evolution **71**
6.1 Introduction 71
6.2 Adding learning to cellular development 72
6.2.1 The Baldwin Effect 72
6.2.2 Developmental Learning 74
6.2.3 Comparative Goals 75
6.2.4 Implementation of the Hebbian Learning 75
6.2.5 Operationalizing Developmental learning 77
6.3 Simulation of different possible combinations 79
6.3.1 Strategic Mutation 79
6.3.2 Parity 79
6.3.3 Symmetry 81
6.4 Conclusion 81

7 The Switch Learning **83**
7.1 Specification of the learning algorithm 83
7.2 The Switch learning algorithm 83
7.3 Simulation with the basic Switch learning algorithm 86
7.4 Extension of the Switch learning Algorithm 88
7.5 Simulation with the extended Switch learning algorithm 89
7.6 Simulation with the Genetic Algorithm 91
7.7 Conclusion 92

8 The mixed parallel GA **93**
8.1 State of the art of Parallel Genetic Algorithms 93
8.2 The mixed parallel GA 93
8.3 A super-linear speed-up 95

9 Modularity **97**
9.1 Adding modularity to Cellular Encoding 97
9.2 Experiments with modularity 99
9.3 Conclusion 101

10 Conclusion: GA + CE = Genetic Programming of neural networks **103**
10.1 A hierarchy of Genetic Language . . . 104
10.2 Cellular Encoding Versus LISP . . . 106
10.3 Evolving the genetic language itself . . . 108
10.4 A successful marriage . . . 108

A Personnal Bibliography **115**
A.1 Reviewed articles . . . 115
A.2 Patent . . . 115
A.3 Conference with published proceedings . . . 115
A.4 National Conference with published proceedings . . . 116
A.5 Research report . . . 116
A.6 Conferences without proceedings . . . 116

B Cellular Encoding is a graph grammar **117**
B.1 Introduction . . . 117
B.2 Cellular Encoding presented as a graph grammar . . . 117
B.3 Encoding a Grammar into a set of trees . . . 118
B.4 Equivalence with the algebraic approach to graph grammar . . . 120
B.5 Conclusion . . . 123

C Technical Complements **127**
C.1 Small example of compilation . . . 127
C.2 The other macro program symbols . . . 131
C.3 Registers of the cell . . . 142
C.4 Syntax of the micro coding of cellular operators . . . 143
C.5 List of program symbols and their microcode . . . 143
C.6 Syntax of the parse trees that are used. . . . 145
C.7 Predefinitions . . . 146
C.7.1 Arithmetic operators. . . . 146
C.7.2 Reading and writing in an array . . . 147
C.7.3 Recursive macro program symbol for the manipulatin of the environmnet . . . 147
C.8 Rules for rewriting nodes of the parse tree . . . 148
C.9 Library of functions for handling arrays. . . . 149
C.10 Trained neural networks . . . 149
C.11 The extended switch learning . . . 149

Chapter 1

Introduction to Genetic Neural Networks

1.1 Genetic Algorithms

Genetic Algorithms (GA) aim at finding the minimum of a mapping f, defined on fixed length bit strings. GA use a population of bit strings, that evolve over many generations, until a sufficiently good solution has been found, or the allocated time has elapsed. The bit strings are called chromosomes. The evolution of the population is determined by three steps.

- During the selection step, individuals are reproduced, with a higher probability if their image by f is small, with respect to the other individuals.
- During the cross over step, individuals are mated, and exchange pieces of their bit string.
- During the mutation step, each individual changes its bit string with a small probability.

A theorem called "shema theorem" ensures a global improvement of the population, from one generation to the other, provided that an hypothesis called "building block hypothesis" is verified. Genetic algorithms are robust, they can be applied to any optimization problem. The only requirement is to be able to encode the solutions on bit strings. They are easy to parallelize. A parallel GA often has a super linear speed up. We have presented the basic GA, that dates back to 1975, when John Holland, the father of GA, edited a book called "Adaptation in natural and artificial environment". Since 1975, a great amount of work has been done on this subject, in USA and in Germany (Evolutionnary Strategies). These research does mathematical analysis, experiments various kind of cross-over, mutation and selection, explores other operators inspired from biology, and other coding methods than the bit string. Next section presents a case where labeled trees are used, instead of bit strings.

1.2 Genetic Programming

John Koza [16] used Genetic Algorithms to evolve LISP programs. LISP is a language for programming with functions. A LISP program is an S expression that is a rooted, labeled, tree with ordered branches. Labels are symbol of LISP functions. Leaves are labeled with constants, or program inputs. The output of the program is the value computed from the S expression. During the recombination between two mother and father tree, a sub tree is cut from the father tree, and replaces a sub tree from the mother tree. Koza does not use mutation. He thinks that mutation is already implemented by the recombination, when the two sub trees that are exchanged are leaves. John Koza uses the above GA, without the mutation step. He applies the GA to labeled trees instead of bit strings. The search space is the space of all possible labeled trees. Koza wrote a first book in which he shows that genetic programming allows to automatically generate programs that solve a great number of problems in various domains. He wrote a second book [17] where he used LISP S-expressions that allow automatic definition of function, and reusability. He showed that the use of functions allows to solve more complex problems. We used the same technic of coding into labeled trees. We will see in the conclusion that the method for generating neural networks, proposed in this thesis, can be named genetic programming of neural networks, not only because the GA handles the same data structure of labeled trees.

1.3 Neural networks

Genetic Algorithms can be understood has a model for learning. In this model, the information is distributed on all the individuals. Neural networks represent another model of learning with a different flavor. A neural network is an oriented graph of cells. Each cell computes an activity from the activities of the neighbors connected to its input. Each cell propagates its activity to the neighbors connected to its output. The computation done by a cell is to apply a function called sigmoid to a weighted sum of its inputs. The weights represent the information that is contained in the neural network. We define a mapping that is computed by the neural network. This definition differs, following the particular model of neural network. The aim of learning is to make the neural net compute a particular function. Most often, this is done using a progressive refinment of the weights. During one iteration, the weights are slightly modified by adding a small quantity, so as to minimize the gradient of an error. Recently, some research aim at also optimizing the topology of the neural network. This research still use a progressive refinment. During one iteration, a neuron or a connection is added or suppressed. Chang [18] wrote a book on this type of methods, so called "constructivist method". Although constructivist algorithm optimize the architecture, they still use a progressive refinment, starting from an initial solution. Constructivist algorithm cannot generate neural networks with a modular structure, that reflects the structure of the problem to be solved. They cannot generate a neural network composed of different functional units.

1.4 Genetic Neural Networks

There is now a relatively large number of papers that deal with various combinations of genetic algorithms (GA) and neural networks. A survey of this work is given by Schaffer, Whitley and Eshelman [28] in an introduction to the proceeding of a workshop on *Combinations of Genetic algorithms and Neural Networks.* Genetic Algorithms have been used to do weight training for supervised learning and for reinforcement learning applications. Genetic algorithms have also been used to select training data and to interpret the output behavior of neural networks. Finally, GAs have also been applied to the problem of finding neural network architectures. This is the problem we are interested in. An architecture specification indicates how many hidden units a network should have and how these units should be connected.

In order to apply a GA to the problem of finding good neural network architectures, there is nothing but one problem to solve: how to code an architecture on the structures called chromosomes, that can be manipulated by the GA. Three classes of methods exist:

direct encoding A graph data structure is encoded. In [32] the connectivity matrix of the network is directly encoded. This gives very long chromosomes of size n^2 for a network including n neurons. Because of this scalability problem, only small networks (four hidden units) have been found. Moreover, making cross-over directly on the hardware of the connections can produce non-functional offsprings. It is known as the structural/functional problem [32]. The solution is to encode on the chromosome a collection of abstract features more likely to provide building blocks. There must be some abstraction in the representation.

parametrized encoding A list of parameters is encoded. In [9] the parameters describe the number of layers, the size of the layers, and how layers are interconnected. This method is more abstract. It could provide big nets with small chromosomes, but only within a restrictive range of architectures: layered architectures. It cannot find modular architectures, where modularity is defined in section 3.4.

grammatical encoding A rewriting grammar is encoded. The grammar is interpreted in a recursive manner, and allows to generate a family of related neural networks. In [14] Kitano begins to implement scalability, abstraction and modularity. In his method, the chromosome encodes a matrix grammar. A rule of this grammar rewrites a 1×1 matrix into a 2×2 matrix. The application of the rules is as follows: One starts with the 1×1 matrix denoted M_0 whose single character is the axiom of the grammar. During the first step, using a rule that rewrites the axiom, this matrix is transformed into a 2×2 matrix of characters denoted M_1. During the second step each character of M_1 is rewritten into a 2×2 matrix. This yields a 4×4 matrix M_2. The process goes on, until the $2^k\times2^k$ matrix M_k where 2^k is greater than n. The symbol n refers to the desired number of units of the target neural network. Extracting from M_k a $n\times n$ matrix, and mapping each character to either 1 or 0, one get a $n\times n$ matrix whose elements are 0 or 1. This matrix is the connectivity matrix of the encoded neural net. This grammar encoding can code big nets with short chromosomes, without imposing the architecture to have a particular shape.

The last method clearly improves upon the two previous. By optimizing grammars that generate network architectures instead of directly optimizing architectures this method hopes to achieve better scalability, and in some sense, reusability of network architectures. In other words, the goal of this method is to find rules for generating networks which will be useful for defining architectures for some general class of problems. In particular, this would allow developers to define grammars on smaller problems and then use theses grammars as building blocks when attacking larger problems.

Nevertheless, there are certain problems with Kitano's representation scheme. Kitano uses a matrix grammar. An $m\times m$ matrix must be developed for a network of n neurons, where m is the smallest power of 2 bigger than n. In order to get an acyclic graph for a feed forward neural network, one must consider only the upper right triangle, which also decreases the efficiency of the encoding.

Mjolness [22] defines a recursive equation for a matrix from which he computes a family of integer matrices and then a family of weighted neural nets; Mjolness also uses a matrix grammar in this sense. The approach is similar to that used by Kitano. Unlike Kitano, Mjolness is not restricted to only using matrices of size 2^k; it is not clear, however, what is the range of possible sizes. Also, the size of the neural network has to be determined in advance.

1.5 Overview

Kitano and Mjolness schemes are both based on a matrix grammar, their grammar replace an element of a matrix by a 2×2 matrix. In this thesis, we propose an encoding based on a graph grammar. It is called cellular encoding. A cellular encoding directly develops a family of neural nets and avoids the need to go through the matrix representation. Instead of a matrix, the object on which the grammatical rules are applied is the cell of a network. Each cell has a copy of the code. Each cell reads the code at a different position. Depending on what it reads, a cell can divide, change some internal parameters, and finally become a neuron. It is both more natural and more efficient to act development at the level of cells rather than on elements of a connection matrix. The resulting language can describe networks in a clear and compact way, and the representation can be readily recombined by the GA.

In the first part of this thesis, we will demonstrate the interest of cellular encoding in itself, as a coding scheme. We will present cellular encoding, show that it is a parallel graph grammar, and demonstrates a list of properties. With the use of GA, despite the fundamental importance of the encoding, there has not yet been any attempts to establish a description of which theoretical properties this encoding should have in the particular case of neural network optimization. We will show that it is possible to define verifiable properties of good neural network encodings. The representation on the chromosome is abstract and compact. Any chromosome develop a valid phenotype. The developmental process gives modular and interpretable architectures, with a powerful scalability property. These theoretical properties show that cellular encoding suits the GA. We will then describe a Pascal compiler that produces the cellular code of a neural network that computes what is specified in the Pascal Program. This will show that cellular encoding is truly a neural network machine language. The compiler in itself presents some interests. It can be considered as a tool for the design of huge neural networks. It can be used as an automatic parallelizer, for programs written with a divide and conquer strategy. It can also compile naive hybrid systems.

In the second part of this thesis, we will demonstrate the efficiency of cellular encoding with respect to the GA. Simulation on hard problem that had not been solved before, will be reported. The GA is used to find both the Boolean weights and the architecture for neural networks that learn Boolean functions. The GA alone can find a solution neural networks, so there is no need to do a backpropagation to optimize the weights and time is saved. Nevertheless, we still propose to use learning to enhance the method, but the goal of the learning and the way it is combined are differents. The GA is not used to find architectures adapted to a particular learning algorithm. Instead, learning is used to speed up the GA. We compare the speed up obtained with different possible ways of combining learning and the GA. The Genetic Algorithm allows a global search of the search space, and find a set of sub-neural networks that the neural-networks solution to the problem should include. The learning tunes the draft neural networks produced by the GA, and thus can speed-up the process. We show that a new combination between GA and learning that uses the Baldwin effect is more efficient than the Lamarckian way of using a learning. This combination mode is called developmental learning. We will then propose a learning method called switch learning that is adapted to the GA. It is a fast learning that changes a few boolean weights for deep neural networks, which are almost correct. The parallel GA that we have programmed will be reported. An example of modular search that could be done using all the findings of the preceding chapters (developmental learning, switch learning, parallel GA) will be given.

Genetic programming uses the GA to evolve LISP computer programs. In the conclusion we compare Genetic synthesis of neural networks using cellular encoding and genetic programming.

Part I: Cellular Encoding

In the first part of this thesis, we present cellular encoding and study its properties.

Chapter 2

Presentation of Cellular Encoding

2.1 Basic Cellular Encoding

Cellular encoding is a method for encoding families of similarly structured neural networks. In this chapter we present the basic cellular encoding. Along the next chapters, we will extend the alphabet of the code, when it is needed. The basic cellular encoding will be used by the genetic algorithm to synthesize neural networks, in the second part of this thesis. Along the next chapters, we will consider extensions of cellular encoding for other purpose than the GA.

The cellular code is represented as a *grammar tree* with ordered branches whose nodes are labeled with name of program symbols. The reader must not make the confusion between grammar tree and tree grammar. Grammar tree means a grammar encoded as a tree, whereas tree grammar means a grammar that rewrites trees. A cell is a node of an oriented *network graph* with ordered connections. Each cell carries a duplicate copy of the cellular code (i.e., the grammar tree) and has an internal reading head that reads from the grammar tree. Typically, each cell reads from the grammar tree at a different position. The character symbols represent instructions for cell development that act on the cell or on connections of the cell. During a step of the development process, a cell executes the instruction referenced by the symbol it reads, and moves its reading head down in the tree. One can draw an analogy between a cell and a Turing machine. The cell reads from a tree instead of a tape and the cell is capable of duplicating itself; but both execute instructions by moving the reading head in a manner dictated by the symbol that is read. In this section we shall refer to the grammar tree as a *program* and each character as a *program-symbol*.

A cell also manages a set of internal registers, some of which are used during development, while others determine the weights and thresholds of the final neural net. The link register is used to refer to one of possibly several fan-in connections (i.e., links) into a cell.

Consider the problem of finding the neural net for the exclusive OR (XOR) function. Neurons can have thresholds of 0 or 1. Connections can be weighted -1 or $+1$. If the weighted sum of its input is strictly greater than its threshold, the neuron outputs 1, else it outputs 0. The inputs of the neural net are 0 or 1.

The development of a neural net starts with a single cell called the *ancestor cell* connected to an input pointer cell and an output pointer cell. Consider the starting network on the right half of figure 2.1 and the Cellular Encoding depicted on the left half of figure 2.1. At the starting step 0 the reading head of the ancestor cell is positioned on the root of the tree as shown by the arrow connecting the two. Its registers are initialized with default values. For example, its threshold is set to 0. As this cell repeatedly divides it gives birth to all the other cells that will eventually become a neuron and make up the neural network. A cell is said to become a neuron when it looses its reading-head. The input and output pointer cells to which the ancestor is linked (indicated by boxes in the figure) do not execute any program-symbol. Rather, at the end of the development process, the upper pointer cell is connected to the set of input units, while the lower pointer cell is connected to the set of output units. These input and output units are created during the development, they are not added independently at the end. After development is complete, the pointer cells can be deleted. For example, in figure 2.5, the final decoded neural net has two input units labeled "a" and "c", and one output unit labeled "d".

- A division-program symbol creates two cells from one. In a *sequential division* (denoted by `SEQ`) the first child cell inherits the input links, the second child cell inherits the output links of the parent cell. The first child connects to the second with weight 1. The link is oriented from the first child to the second child. Since there are two child cells, a division program-symbol must label nodes of arity two. The first child moves its reading head to the left subtree and the second child moves its reading head to the right subtree. This is illustrated in steps 1 and 3.

- Parallel division (denoted by `PAR`) is a second kind of division program symbol. Both child cells inherit the input and output links from the parent cell (in step 2 and step 6). Finally, when a cell divides, the values of the internal registers of the parent cell are copied in the child cells.

- The *ending-program symbol* denoted `END` causes a cell to lose its reading head and become a finished neuron. Since the cell does not read any subtree of the current node, there is no subtree to the node labeled by `END`. Therefore `END` labels the leaves of the grammar tree (i.e., nodes of arity 0).

- A value-program symbol modifies the value of an internal register of the cell. The program-symbol `INCBIAS` increments (and `DECBIAS` decrements) the threshold of a cell. The value-program symbol `INCLR` increments (and `DECLR` decrements) the value of the link register, which points to a specific fan-in link or connection. Changing the value of the link register causes it to point to a different fan-in connection. The link register has a default initial value of 1, thus pointing to the leftmost fan-in link. Operations on other connections can be accomplished by first resetting the value of the link register. The program-symbol denoted `VAL+` sets the weight of the input link pointed by the link register to 1, while `VAL-` sets the weight to -1 (see step 7). The program-symbols `VAL+` or `VAL-` do not explicitly indicate to which fan-in connection the corresponding instructions are applied. When `VAL+` or `VAL-` is executed it is applied to the link pointed to by the link register.

- An unary program-symbol `CUT` cuts the link pointed by the link register. This operator modifies the topology by removing a link, whereas the reseting of the link register does not modify the topology.

Operators `INCLR`, `DECLR`, `CUT` are not illustrated, they are not required for the development of a neural net for the XOR problem. The sequence in which cells execute program-symbols is determined as follows: once a cell has executed its program-symbol, it enters a First In First Out (FIFO) queue. The next cell to execute is the head of the FIFO queue. If the cell divides, the child which reads the left subtree enters the FIFO queue first. This order of execution tries to model what would happen if cells were active in parallel. It ensures that a cell cannot be active twice while another cell has not been active at all. In some cases, the final configuration of the network depends on the order in which cells execute their corresponding instructions. For example, in the development of the XOR, performing step 7 before step 6 would produce a neural net with an output unit having two negative weights instead of one, as desired. The waiting program-symbol denoted `WAIT` makes the cell wait for its next rewriting step. `WAIT` is necessary for those cases where the development process must be controlled by generating appropriate delays.

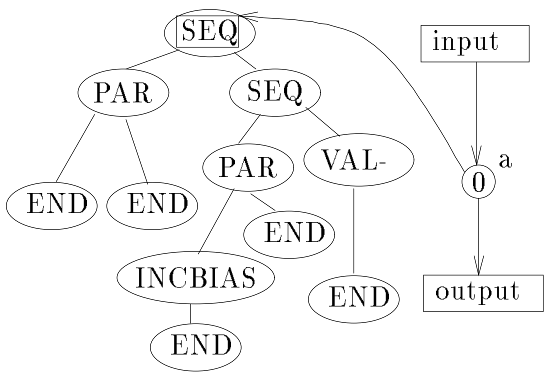

Figure 2.1: Step 0: Development of the neural net for the exclusive-OR (XOR) function in the right half of this figure. The development starts with a single ancestor cell labeled "a" and shown as a circle. The 0 inside the circle indicates the threshold of the ancestor cell is 0. The ancestor cell is connected to the neural net's input pointer cell (box labeled "input") and the neural net's output pointer cell (Box labeled "output") The Cellular Encoding of the neural net for XOR is shown on the left half of this figure. The arrow between the ancestor cell and the symbol `SEQ` of the graph grammar represents the position of the reading head of the ancestor cell. A continuous line indicates a weight 1, and a heavy line a weight -1. By default all the weights are 1.

2.2 Coding of recursivity

Up to this point in our description the grammar tree does not use recursion. (Note that recurrence in the grammar does not imply that there is recurrence in the resulting neural network.) One is able to develop only a single neural network from a grammar tree. But one would like to develop a family of neural networks, which share the same structure, for computing a family of similarly structured problem. This would allow to get a property of scalability. For this purpose, we introduce a recurrent program-symbol denoted **R** which allows a fixed number of loops L. The cell which reads **R** executes the following algorithm:

```
life := life - 1
If (life > 0) reading-head := root of the tree
Else  reading-head := subtree of the current node
```

where *life* is a register of the cell initialized with L in the ancestor cell.

Thus a grammar develops a family of neural networks parametrized by L. The use of a recurrent-program symbol is illustrated figure 2.6 and 2.7. The cellular code in this figure is almost the same as the cellular code of a XOR network. The only difference is that an end-program symbol `END`

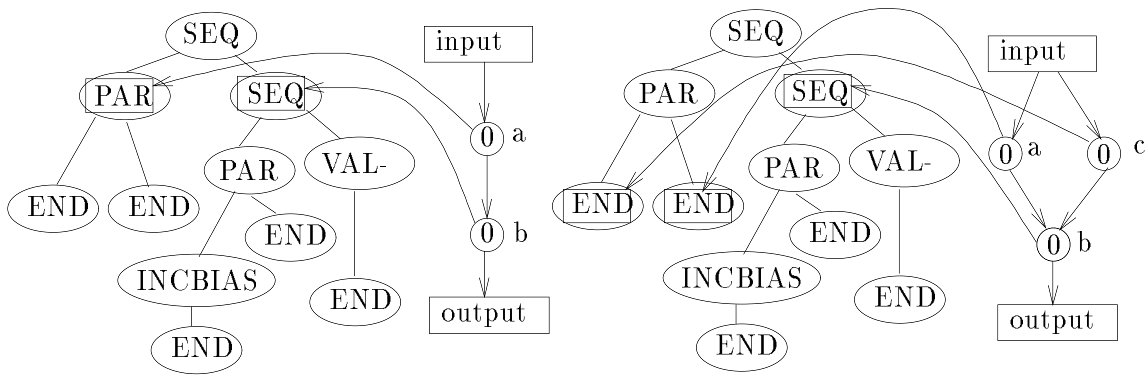

Figure 2.2: On the left: the execution at step 1 of the sequential division `SEQ` pointed to by the ancestor cell "a" of preceding figure causes the ancestor cell "a" to divide into two cells "a" and "b". Cell "a" feeds into cell "b" with weight +1. The reading head of cell "a" now points to the left sub-tree of the Cellular Encoding on the left (the box with `PAR`) and the reading head of new cell "b" points to the right subtree (the box at the second level down with `SEQ`). On the right: The execution of the parallel division `PAR` at step 2 causes the creation of cell "c". Both cell "a" and "c" inherit the input formerly feeding into "a". Both cells "a" and "c" output to cell "b" (The place where "a" formerly sent its output.) The reading head of cell "a" now points to the left sub-tree (`END`) and the reading head of the new cell "c" now points to the right sub-tree (which also has an `END`).

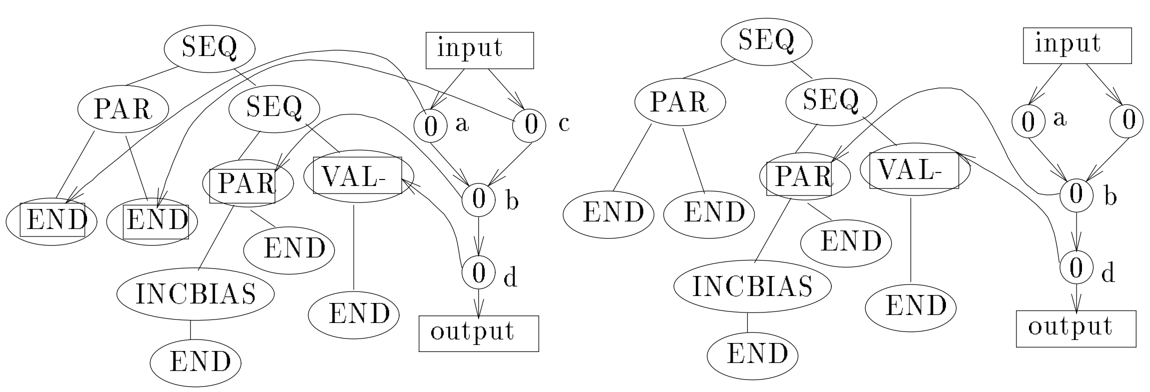

Figure 2.3: On the left: the execution at step 3 of the sequential division `SEQ` pointed to by the cell "b" of preceding figure causes the cell "b" to divide into two cells "b" and "d". Cell "b" feeds into cell "d" with weight +1. The reading head of cell "b" now points to the left sub-tree (the box at the third level with `PAR`) and the reading head of new cell "d" points to the right subtree (the box at the third level down with `VAL-`). On the right: The execution of the two end-program symbols. The `END`'s causes the cells "a" and "c" to lose their reading head and become finished neurons. Since there are two `END`'s, it takes two time steps, one time step for each `END`

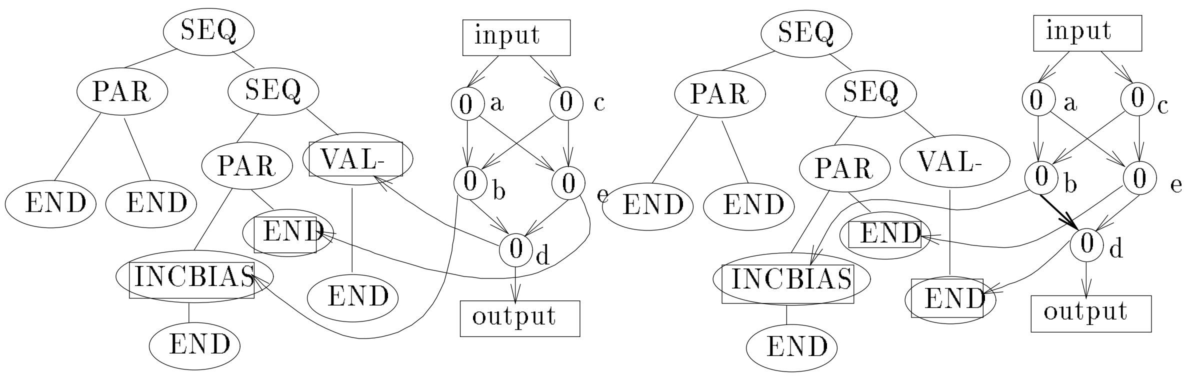

Figure 2.4: On the left: the execution of the parallel division "**P**" at step 6 causes the creation of cell "e". Both cell "b" and "e" inherit the input from cell "a" and "c" formerly feeding into "b". Both cells "b" and "e" send the output to cell "d" (The place where "a" formerly sent its output.) The reading head of cell "b" now points to the left sub-tree (`INCBIAS`) and the reading head of the new cell "e" now points to the right sub-tree (which has an "**E**"). On the right: The cell "c" executes the value-program symbol `SET-`. The link register is one (the default value) it points the left-most link. The action of `SET-` is to set the weight of the first link to -1. The dashed line is used to indicate a -1 weight. After the execution of `SET-` the cell "c" goes to read the next program symbol which is an `END` located on the fourth level of the tree.

has been replaced by a recurrent-program symbol `REC`. The resulting cellular code is now able to develop a neural net for the parity function with an arbitrary large number of inputs, by assembling copies of a XOR sub-network. In figure 2.7 the network for parity of three inputs is shown. This implementation of the recurrence allows precise control of the growth process. The development is not stopped when the network size reaches a predetermined limit, but when the code has been read exactly L times through. The number L parametrizes the structure of the neural networks.

2.3 Comparison with Kitano 's scheme

Both Kitano's scheme and CE encodes a grammar. Kitano has encoded a matrix grammar. CE encodes a graph grammar. Instead of rewriting matrix, the grammar of CE rewrites neuronal cells. In fact, rewriting on cells is like defining a "neural network machine language". This language can describe network architectures as well as weight patterns in a natural and clear way. During the developmental process, instead of a global rewriting of the connectivity matrix, each node of the graph is processed separately, one after the other. The final product of the rewriting is exactly the encoded neural network and there is no need for an additional process. On the contrary, Kitano's developmental process produces a raw matrix of characters, which needs further processing with some artificial conventions before yielding a neural network. We would like here to stress the advantage of CE upon Kitano's scheme.

- In Kitano's scheme, the same piece of code that represents a subset of rules can be used repeatedly during the decoding phase to produce the same pattern of connections at different places in the neural net. Still, although there can be repeated patterns in the connectivity

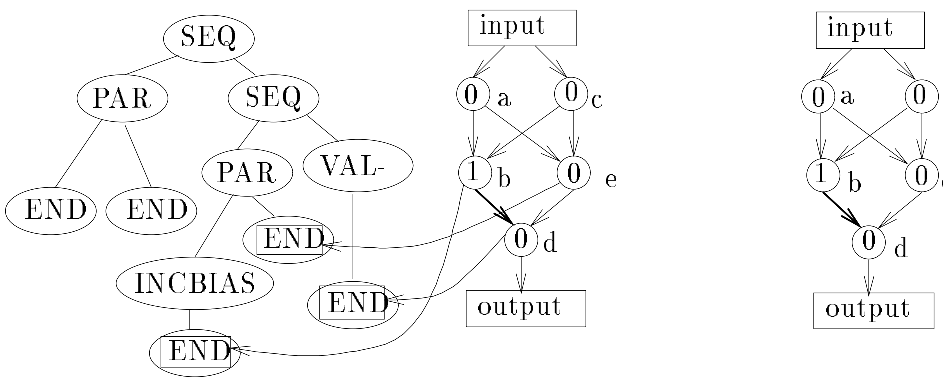

Figure 2.5: On the left: Neuron "b" executes the value-program symbol `INCBIAS`. The action of `INCBIAS` is to increase the threshold of cell "b" by 1. After execution, the threshold of cell "b" is 1, and the reading head of cell "b" points to an `END` at the fifth level of the tree. On the right: The last tree steps consist in executing three end-program symbols. The three `END`'s cause the neural cells "b", "e" and "d" to lose their reading head and become finished neurons. Since there are now only finished neurons, the development is finished. The final neural net has two input units "a" and "c", and one output unit "d". The neuron "a" is the first input unit, because the link from the input pointer cell to "a" is the first output link of the input pointer cell. The neuron "c" is the second input unit, because the link from the input pointer cell to "a" is the second output link of the input pointer cell

matrix, repeated subnetworks do not appear in Kitano's experimental results. A theoretical property of modularity will be proved for CE. And the network generated by the GA will have a clear modular structure.

- Kitano characterizes scalability as the ability of the GA to find big neural networks for a parametrized problem of big size. However, its goodness is tested on an atypical case: a degenerated version of the encoder/decoder problem where the number of neurons in the hidden layer is bigger than the number of input or output neurons. In the original problem, the number of hidden neurons must be the logarithm of the number of input neurons and that is the point that makes the problem challenging. With CE, a theoretical property of scalability is proved. Simulation with CE show that a fixed size code can develop an arbitrary large neural network to solve a boolean problem of arbitrary large size.

- Kitano's encoding scheme requires artificial conventions that very likely decrease the compactness of the code. An $m \times m$ matrix must be developed for a network of n neurons, where m is the smallest power of 2 bigger than n. In the worst case, $n = m/2 + 1$, and extracting an $n \times n$ matrix, one uses a fraction that is $(0.5 - 1/m)^2$ of the bits, which is approximately 25% of the bits if m is large. We do not say that the neuron numbers are limited to power of 2. Only the range of matrix size is limited to power of 2. Nevertheless, the above computation shows that a large matrix will be developed and only a small part of it will be used. In order to get an acyclic graph for a feed-forward network, one must consider only the upper right triangle.

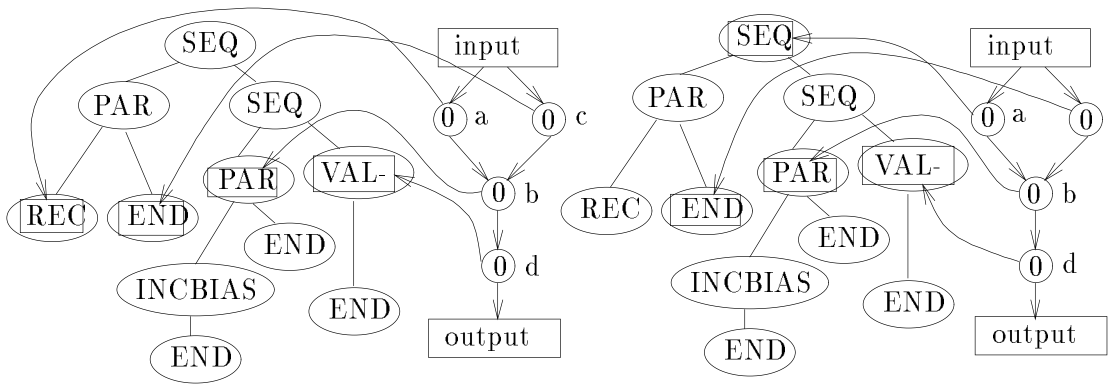

Figure 2.6: A recurrent developmental program for a neural network that computes the even parity for three binary inputs. The life of the ancestor cell is 2. The first three steps are the same as those shown in the preceding figures. During the fourth step, cell "a" executes a recurrent-program symbol, its reading head points now to the label (`SEQ`) on the root of the grammar-tree, as if we were back in the beginning of the development at step 0. The life of "a" has been decremented and is now equal to 1. Cell "a" will give birth to a second XOR network, whose outputs will be sent to the child cells generated by cell "b".

Step 22

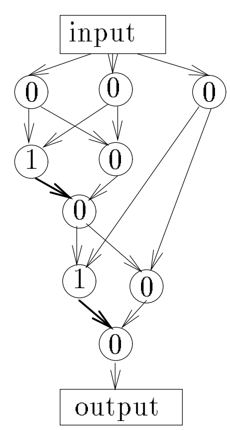

Figure 2.7: The final decoded neural network that computes the even parity for three binary inputs. The life of the ancestor cell is 2. The total number of steps is 22 since the tree is executed 2 times. One can clearly see the two copies of the XOR network. With an initial life of L we would have developed a neural network for the $L+1$ input parity problem, with L copies of the XOR network.

- With Kitano's scheme the grammar is encoded as a set of rules. Grammar produced by the GA are not clean. They may have many rules that rewrite the same character, and rules that are not used. Only a small percentage of the coded rules are used and the efficiency is rather poor. In CE, the grammar is encoded as a rooted, point labeled tree with ordered branches, where the labels refer to program-symbol. It ensures that the GA always produces deterministic grammars where all the rewriting rules are used. Moreover Fig. B.2 shows that trees can be more compact than rules.

- With Cellular Encoding, special attributes allow to encode recursive grammars that produce infinite networks. Instead of stopping the development of the network when the size of the connectivity matrix reaches a given bound, a simple mechanism allows to apply the grammar rules for a fixed number of loops L. Thus, a grammar can develop a family of neural networks parametrized by L.

Chapter 3

Properties of Cellular Encoding

3.1 Introduction

It is now recognized that the key point in successful application of Genetic Algorithms (GA) to an optimization problem, is the scheme used to encode solutions on the structures manipulated by the GA. Despite the fundamental importance of the encoding, there has not yet been any attempt to establish a description of which theoretical properties this encoding should have in the particular case of neural network optimization. We will show that it is possible to define verifiable properties of good neural network encodings. Each previous approach focussed on a particular aspect. Several encoding have been tried, and simulation was reported with an intuitive explanation of the results. One yet has to find a complete set of theoretical properties. In this chapter, seven theoretical and verifiable properties that rate an encoding of neural networks are defined. The properties are: completeness, compactness, closure, modularity, scalability, power of expression, abstraction. The Cellular Encoding (CE) is shown to check the seven properties. Last, it will be shown that cellular encoding is a graph grammar.

There does not exist a single CE but plenty of them. There are two features needed to define a CE: an initial graph of cells, and a set of program-symbols. The figure 2.1 describes the initial graph of cell, before the development starts. It contains a single ancestor cell, and two pointer cells. In this chapter we will use two kinds of initial graphs of cells The ancestor cell of the initial graph can have a recurrent link from its output to its input. This recurrent link is used to develop recurrent networks. The initial network graph without the recurrent link is noted $ACYC$, and the one with a recurrent link, CYC.

A particular CE with r program-symbols is determined by a $r+1-$tuple of symbols. The first symbol refers to the initial network graph and the last r symbols refer to the program-symbols. In the notation, the first symbol will be slanted. Along the next sections, it will be shown that increasing the number of program-symbol allows to achieve a growing set of properties. We will reintroduce the same program-symbols defined in chapter 2, and will give them a more precise definition. Appendix **??** gives a list of all the program-symbols used in this thesis.

3.2 Architecture encoding

In this section, program-symbols that develop the architecture are studied. An architecture is a directed graph. Each vertex stands for a neuron. The architecture includes also two ordered set of neurons. These lists specify the input and output neurons. First a minimal set of three program-symbols encoding any architecture is presented.

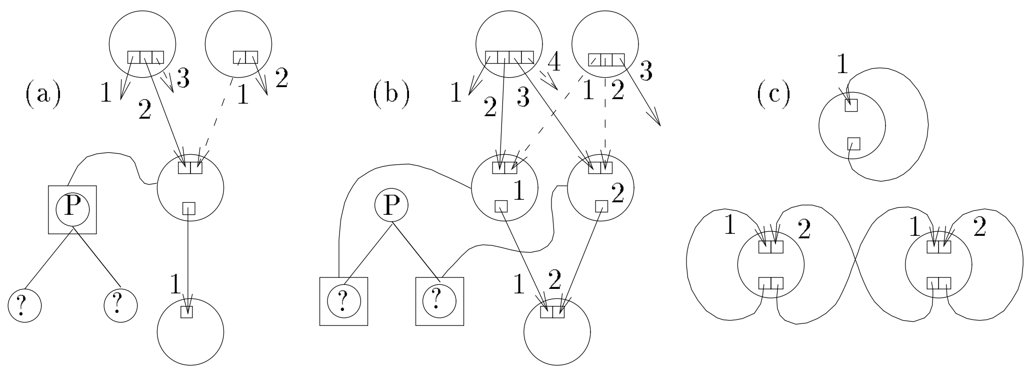

Figure 3.1: The parallel division. (a) Before the execution, (b) After the execution, (c) The case of a recurrent link. Weight +1 is represented by — and weight -1 by - - - .

The first program-symbol is the parallel division noted PAR. It is a 2-ary program-symbol. In the parallel division both child cells inherit the input and output links from their father cell. If the parent cell has a recurrent link, the two children connect to each other and to themselves. Fig. 3.1 shows how the links have to be reordered once the new cell has been added. The second program-symbol noted CUT cuts an input link. It has an integer argument which specifies the number of the input link to cut. The third program-symbol noted END makes the cell lose its reading head and become a terminal neuron.

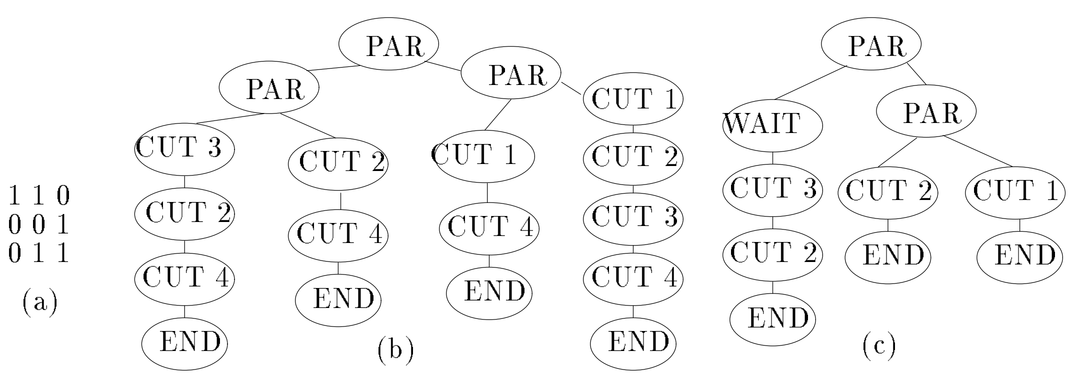

Figure 3.2: Encoding of an arbitrary recurrent network. (a) The target connectivity matrix, (b) Code that uses two program-symbols PAR CUT, (c) Code that uses three program-symbols PAR CUT WAIT.

Fig. 3.2 shows how to encode an arbitrary graph. Since all the leaves and only the leaves are labeled with END, the program-symbol END will not be represented in the next figures. The development of a given graph having n nodes is done in two stages. During the first stage, a fully connected graph of m nodes is developed, where $m = 2^{\lceil \log_2(n) \rceil}$ and $\lceil x \rceil$ is the ceiling function. This is done with a binary tree of depth $\lceil \log_2(n) \rceil$ labeled with PAR. During the second stage, each cell cuts its supernumerary input links. All the links to the supernumerary $n-m$ cells are cut. The final graph has n nodes, connected as desired. The following property is convenient for the formalization:

Property 1 (completeness) *An encoding scheme is complete with respect to a set of architectures (resp. neural networks) if any element of the set can be encoded.*

The preceding construction yields:

Proposition 1 { *CYC*, `END`, `PAR`, `CUT` } *is complete with respect to the set of all architectures.*

Because of the program-symbol `CUT` , the topology of the final network depends on which cell is rewritten first. In the preceding construction, the cells start to cut their input links only when the growth of the net is finished. This is why a net which size is an exact power of two must be developed. This development can be avoided by means of an unary waiting program-symbol `WAIT`, which makes the cell wait for the next rewriting step. The program-symbol `WAIT` is necessary for a rewriting order to be encoded by generating appropriate delays. Fig. 3.2 shows how `WAIT` can be used to encode a neural network with a shorter chromosome. There is no need to create more cells than the final number of cells. Every cell starts to cut at the same time. The cells that have finished their division earlier are just delayed with the program-symbol `WAIT`. Clearly, this construction makes the code shorter. Let us define the property of compactness as follows:

Property 2 (compactness) *An architecture encoding scheme $\mathcal{E}$ is said to be topologically more compact than another encoding $\mathcal{F}$ if every architecture encoded by $\mathcal{F}$ with a code of size s, can also be encoded by $\mathcal{E}$ with a code of size $O(s)$. If the schemes also encodes weights, $\mathcal{E}$ is said to be functionally more compact than $\mathcal{F}$ if for every neural network N encoded by $\mathcal{F}$ with a code of size s, there exists a neural network encoded by $\mathcal{E}$ with a code of size $O(s)$ that implements the function associated to N.*

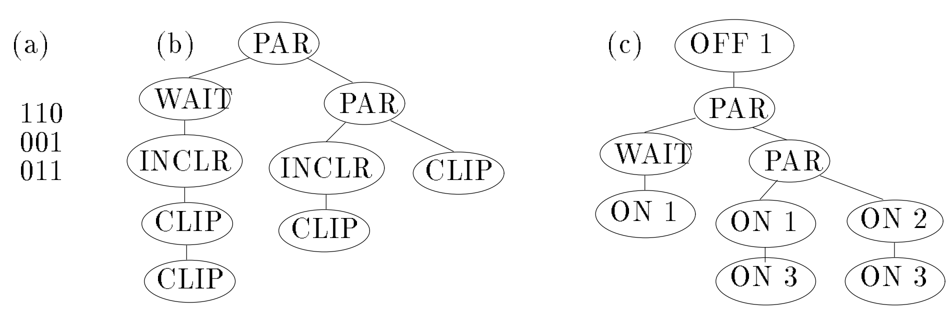

Figure 3.3: Compact encoding of an arbitrary recurrent network. (a) The target connectivity matrix, (b) Compactness with respect to $\mathcal{C}1$, (c) Compactness with respect to $\mathcal{C}2$.

The size of a code is the number of bits necessary to store it. Since less than 31 program-symbols will be used, 5 bits suffice to encode each program-symbol plus a NULL character. Hence, the size of the code needed to store a tree of α nodes is $5\alpha+1$. We want to show compactness with respect to the classical representations of graphs. First consider the direct encoding $\mathcal{C}1$ of the connectivity matrix. Recall that it produces a code of size n^2 for a network of n neurons. Provide the reading cell with an internal register called link register that contains a link number. Define the unary program-symbol `INCLR` incrementing the link register and the unary program-symbol `CLIP` cutting the link pointed by the link register. As shown if fig. 3.3, each cell of the fully connected graph can select its subset of connections with the execution of less than n `CLIP` alternated with n `INCLR`. Since the fully connected graph has n cells, the size of the total code is $O(n^2)$. The following holds:

Proposition 2 *{ CYC,* `END`*,* `CLIP`*,* `INCLR`*,* `PAR`*,* `WAIT` *} is topologically more compact than $\mathcal{C}1$.*

Since the program-symbol `CUT` can be implemented using `CLIP`, it will not be used any more, except for the sake of clarity, in the figures. `CLIP` will be denoted as a `CUT` with no arguments.

Second, consider the representation $\mathcal{C}2$ where the list of connections is encoded. The neuron numbering can be encoded in $\log(n)$ bits. For each connection, the numbers of the two connected neurons are specified. Therefore $\mathcal{C}2$ produces an encoding of size $O(l\log(n))$ for a network with l connections. This encoding is interesting when there are few connections. Assume a link has two states: `on/off`. Like cells, links also have registers. This state is stored in a register of the link called `state`. When a link is duplicated during a cell division, the register of the link are also duplicated. At the end of the rewriting, the links with states `off` are removed. Define the program-symbol `ON` that turns `on` the state of the link specified by an integer argument, and its counterpart `OFF`. Fig. 3.3 (c) shows how to encode an arbitrary network with the program-symbols `ON` and `OFF`. Let the default value for the recurrent link in the initial graph be `on`. The program-symbol `OFF 1` labeling the root of the grammar tree sets the state of the recurrent link to `off`. The program-symbols `PAR` develop a fully connected graph where all the links are `off`. Last, each cell c builds its set of l_c connections by applying l_c program-symbols `ON` on the numbers of the desired neighbors. Since each of these numbers are coded with $\log(n)$ bits, the size of the total code is $O(l\log(n))$. The following holds:

Proposition 3 *{ CYC,* `END`*,* `ON`*,* `OFF`*,* `PAR`*,* `WAIT` *} is topologically more compact than $\mathcal{C}2$.*

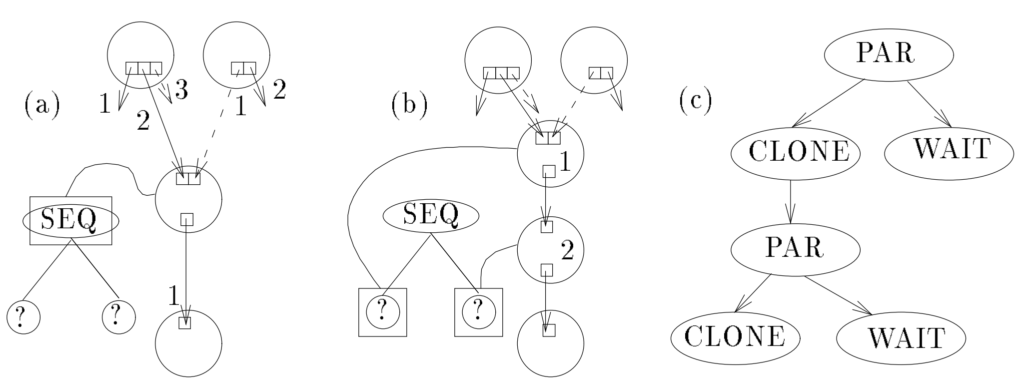

Figure 3.4: How to encode efficiently a layered neural network. (a) Before the sequential division, (b) After the sequential division, (c) The code of a layer of 7 cells using the clone program-symbol. Weight +1 is represented by — and weight -1 by - - - .

Last, consider the scheme $\mathcal{C}3$ that encodes layered neural networks, with complete interconnections between layers. $\mathcal{C}3$ specifies the number of neurons in each layer. Define two other division program-symbols. The sequential division noted `SEQ` makes the first child cell inherit the input links of the father cell, the second child inherits the output links, and the first child connect to the second with a link whose weight is +1 and state is `on`. An example is shown in fig. 3.4 (a) (b). The clone division denoted `CLONE` is like `PAR` except that the two child cells place their reading head on the same node of the grammar tree. Since there is only one subtree to the nodes labeled with `CLONE` , this program-symbol is unary. It is called clone because the two child cells will execute exactly the same code, and therefore give birth to the same subnetwork of neurons. A given neural network with c layers is developed as follows: Apply $c-1$ times `SEQ` to produce a string of c cells. At this stage, each cell stands for a layer. The rank of the cell in the string corresponds to the layer number. Now each cell standing for layer i must execute n_i times `PAR`, where n_i is the number of cells of the i^{th} layer. As shown in fig. 3.4 (c), a layer of n_i neurons can be developed with a combination of less than $3\log_2(n_i)$ `PAR` and `CLONE` program-symbols. The total size of the code is: $c + 3\sum_{i=1}^{c}\log_2(n_i)$. It is less than three times the size of the code produced by $\mathcal{C}3$. Thus:

Proposition 4 *{ CYC,* `END`*,* `PAR`*,* `SEQ`*,* `CLONE`*,* `WAIT` *} is topologically more compact than $\mathcal{C}3$.*

Let us now define another property.

Property 3 (closure) *An architecture encoding scheme is closed with respect to a set of architectures, if every possible code develops an architecture in this set.*

Whatever the program-symbols are, a given CE is always closed with respect to the set of all architecture. This means that any code generated by the GA can actually develop an architecture. It ensures that whenever the GA generates a chromosome, the GA will be able to develop and evaluate a network, and receive a feed-back.

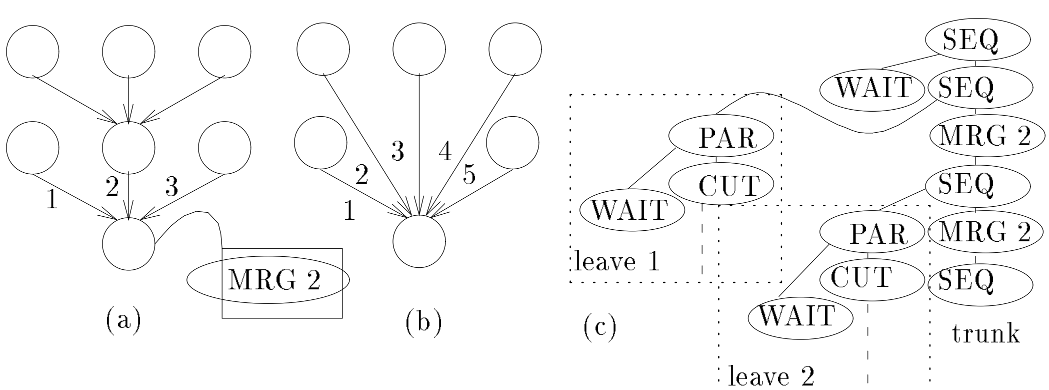

Figure 3.5: The merge program-symbol. (a) Before the application of the program-symbol, (b) After the application, (c) The code of an arbitrary acyclic graph.

Suppose that the architecture one is looking for is known to belong to a particular set. It is possible to make the GA use this a priori knowledge. One must design an encoding scheme that is closed with respect to the set and the GA will automatically place its individuals in the set. For example one can require acyclic architectures in order to be able to do backpropagation. If one starts to rewrite from an acyclic network graph, it will remains acyclic, thus { *ACYC*, `END`, PAR, SEQ, CUT, INCLR, WAIT } is closed with respect to the set of acyclic architectures, but it is not complete with respect to this same set. In order to obtain completeness, the program-symbol merge, denoted MRG, is introduced. The program-symbol MRG has an integer argument i and rewrites a cell in the following manner: let l be the input link which number is i and c the neighbor cell which is connected through l. The program-symbol replaces l by the list of the input links of c, as shown in fig. 3.5 (a) (b). If after this operation, c has no more output links, c is suppressed. Fig. 3.5 (c) shows how to build the code of an arbitrary acyclic graph. In an acyclic graph, it is possible to number the nodes so that each node is connected only to other nodes having a greater number than itself. The proposed code is made of a trunk and n leaves, where n is the number of nodes. At an intermediate step, all the cells of the current network graph are connected to the cell that reads the trunk. Periodically, the cell that reads the trunk divides, and creates a child c that reads leaves i and represents the node of number i. At this stage, all the cells that represent nodes with a number less than i are connected to c. The cell c will read on the i^{th} leave how to prune the supernumerary input links. The corresponding cutting program-symbols are located inside the dashed line in fig. 3.5 (c). Thus:

Proposition 5 { *ACYC*, END, PAR, SEQ, CUT, INCLR, MRG, WAIT } *is closed and complete with respect to the set of acyclic architectures. Adding* ON, OFF *and* CLONE *yields the property of compactness with respect to $\mathcal{C}1$, $\mathcal{C}2$ and $\mathcal{C}3$.*

For each neuron, we want a path from one input unit to this neuron (the output of a neuron not having such a path would constant). One can also require that there exists a path from each neuron to a given output unit (otherwise the neuron would not be used in the computation of the neural net). An architecture that meets these two requirements is a meaningful architecture. Put in another way, an architecture is meaningful if it remains connected, after having suppressed either the input pointer cell, or the output pointer cell.

Let us first show the closure property with respect to the set of meaningful architectures that are also acyclic. The semantic of the program-symbols that prune links must be slightly modified in the following manner: a reading cell can apply one of the program-symbols CUT, OFF, MRG on a link whose state is on only if the number of input links which state is on is strictly greater than one. Take *ACYC* as the initial network graph. Try to suppress either the input pointer cell, or the output pointer cell. The pruned graph is a connected graph that has more than one node. A case analysis shows that the application of any of the program-symbols already defined keeps the network graph connected. Hence the final network graph is connected. And it is the final graph that would have been obtained starting with *ACYC* minus the input pointer or the output pointer cell. From the characterization of meaningful, one deduce that the final architecture is meaningful and:

Proposition 6 { *ACYC*, END, PAR, SEQ, CUT, INCLR, MRG, WAIT } *is closed with respect to the set of meaningful acyclic architectures.*

Now, consider the case of meaningful recurrent architectures. The construction uses the fact that a meaningful recurrent architecture contains a meaningful acyclic architecture. Take the initial network graph *CYC*. Define an order in the cells of the network graph with two rules:

- During a cell division, the child that reads the left subtree is less than the one that reads the right subtree.
- If a cell $c1$ is less than a cell $c2$, any descendant of $c1$ including $c1$ is less than any descendant of $c2$ including $c2$.

Provide each cell c with two input sites, one for the link connected to cells less than c, and one for the link connected to cells greater than or equal to c. When a cell division takes place, the two input sites have to be managed so as to respect their definition. In order to develop a meaningful recurrent architecture, one must respect the preceding restriction in the application of pruning program-symbols with the input site connected to inferior cells.

With `ON`, `OFF` and `CLONE`, one can again derive the property of compactness with respect to $\mathcal{C}1$, $\mathcal{C}2$ and $\mathcal{C}3$. The construction with { `ON`, `OFF` } is difficult because the new semantics does not allow to set all the links to `off`. At each step, for each cell, at least one input link must be `on`. Hopefully, this can be ensured without increasing the size of the code more than the desired complexity.

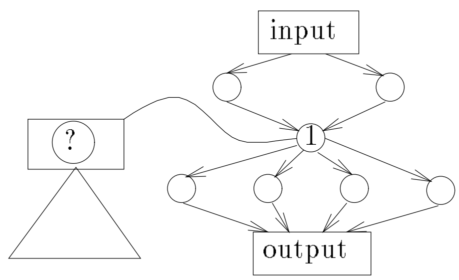

Figure 3.6: Imposing 2 input units and 4 output units. The cell labeled 1 is the ancestor cell.

Last, one wishes to impose the number of input units and output units, one can take the kind of initial network graph shown in fig. 3.6.

3.3 Neural Network Encoding

In this section we describe how to encode a neural network. The architecture, the weights, and the kind of sigmoïds used by each neuron must be encoded. A simple way of encoding weights is to provide an unary program-symbol `VAL` which weights the link pointed by the link register of the cell to the value specified by an argument. This argument can be ± 1, integer or real, depending on the desired neural network. Consider the encoding scheme $\mathcal{C}1$ and $\mathcal{C}2$ defined in the preceding section. The encoding scheme $\mathcal{D}1$ is like $\mathcal{C}1$ except that a list of weight values is provided together with the connectivity matrix. The same holds with $\mathcal{D}2$ and $\mathcal{C}2$. We have the following result:

Proposition 7 { *CYC*, `END`, `PAR`, `SEQ`, `CUT`, `INCLR`, `ON`, `OFF`, `MRG`, `WAIT`, `VAL`, } *is complete with respect to the set of all neural networks, where the units have the same sigmoïd. It is functionally more compact than $\mathcal{D}1$ and $\mathcal{D}2$.*

To show these properties, a two stage development is used. First, the architecture is developed, then the weights are set. From a GA point of view, an encoding using the `VAL` program-symbol presents some drawbacks. The GA forms its initial population with random individuals. As described by Koza in [16], when an program-symbol like `VAL` needs an argument, a random value is generated in a specified range. If there is no mutation, which is the case in Koza's scheme, the possible value of the argument in the future generations are limited to the set of values represented in the initial population. In the case of the program-symbol `VAL`, it means that the possible values of the weights are restricted to this finite set. If a particular weight is needed, and is not available in the current population, the GA will not be able to provide it. Moreover, with the program-symbol `VAL`, there is no possibility to find values with a progressive refinement, through the formation of building blocks of increasing size. A solution to the problem is to use two other program-symbols: `MULT` which multiplies the weight by its argument, and `ADD` which adds its argument to the weight. It is possible to go further in this direction by using a set of four program-symbols: `INC`, `DEC`, `MULT2` and `DIV2`, which respectively increments, decrements, multiplies by two and divides by two the weight of the input link pointed by the link register. These program-symbols are interesting because they have no arguments. If we use only this set without `VAL`, then the formation of weight values will be done by a subtle combination of these four program-symbols. The precision to which weights are coded is not fixed a priori. If the encoding of a number takes b bits with a fixed point format, an encoding of this number can be built with $O(b)$ program-symbols chosen in { `INC`, `DEC`, `MULT2`, `DIV2` }. This allows to have the same property of compactness with respect to $\mathcal{D}1$ and $\mathcal{D}2$ if the weights are coded in a fixed point format.

A similar technique can be used to remove the argument of other program-symbols. For example, if this argument is a link number, the program-symbol can use the value of the link register. In order to keep the property of compactness, one needs the program-symbol `MULT2LR` which multiplies by 2 the value of the link register, and set the value to 0 if it is greater than the number of input links. For a purpose of clarity, in the next figures, program-symbols will still use an integer argument.

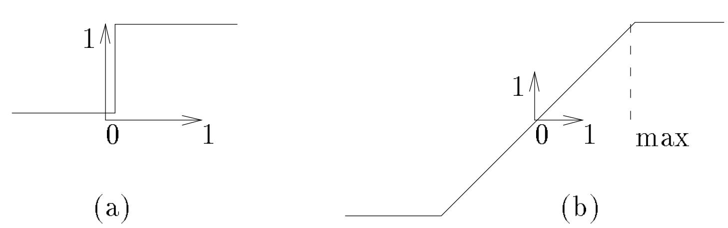

Figure 3.7: Two sigmoïds. (a) the sign function, (b) the bounded linear function. The constant *max* is a parameter of the model.

A neural network is not entirely determined by a weighted architecture. Each neuron may have features on his own, such as a different sigmoïd, or a different learning procedure to adjust its weights. To encode this, an internal register of the cell is attributed to each of these features, and program-symbols that modify the value of these registers are introduced. For example, assume that there is a choice between the two sigmoïds plotted in fig. 3.7. Let a sigmo register stores the type of sigmoïd. An program-symbol `SSIGMO` sets the value of the sigmo register to its argument. When the cell becomes a terminal neuron, its sigmo register contains the number of its sigmoïd.

3.4 Simulation of a Turing machine

In this section, we build the encoding of a neural network that simulates a Turing machine. The transition function of the neurons is the sign function. This, in turns, will enable to obtain interesting other properties. As a first step, we want to achieve the following property.

Property 4 (modularity) *Consider a network N_1 that includes at many different places, a copy of the same subnetwork N_2. The code of N_1 is* **modular** *if it includes the code of N_2, a single time.*

This property tries to formalize the intuitive idea of modularity. Modularity should help to reduce the apparent complexity of an initial network by decomposing it into a set of less complex subnetworks connected by a simple structure. The process is repetitive: subnetworks may themselves be decomposed. Modularity is connected with the reusability problem: the aim is to transform the neural network design into a construction box activity, whereby networks would be built by combination of existing standard elements. Since each module can be separately understood by a human reader, the architectures produced are easily understandable.

In order to include a subnetwork N_2, one must provide a program-symbol that makes the reading head of the cell jump on the subtree defining N_2. One can use a method advocated by Ray in [27] called "Addressing by Template", inspired by molecular biology. Ray writes:

> "In most machine codes, when the IP jumps to another piece of code, the exact numeric address of the target code is specified in the machine code. Consider that in the biological system, by contrast, in order for protein molecule A to interact with protein molecule B, it does not specify the exact coordinate where B is located. Instead molecule A presents a template on its surface which is complementary to some surface on B. Diffusion brings the two together, and the complementary conformations allow them to interact".

Addressing by template is illustrated by the unary program-symbol `JMP`. Each `JMP` program-symbol is followed by a sequence of `NOP` (no operation) program-symbols, of which there are two kinds: `NOP_0` and `NOP_1`. Suppose we have a piece of code with four instructions in this order: `JMP(NOP_0(NOP_0(NOP_1(..))))`. The cell reading the `JMP` will search in the tree the nearest occurrence of the complementary pattern : `NOP_1(NOP_1(NOP_0(..)`. The nearest is defined from the natural topology on a graph where the distance between two vertex is the length of the shortest path that connects them. If the pattern is not found, the `JMP` program-symbol is ignored. If the pattern is found, the cell positions its reading head to the end of the complementary pattern.

The branching can loop. In order to avoid the infinite growth of the network the cell is provided with a life register. The life register of the reading cell in the initial graph is initialized with a value L. If the complementary pattern is found, the `JMP` program-symbol decrements the life register and removes the reading head only if the life register is positive. With this algorithm, the number of loops is bounded by L. A code develops a family of neural networks parametrized by L. The size of the network increases with L.

In order to fulfill the property of modularity, one must provide the possibility to encode separately a subnetwork N_2 from the main network N_1. For this purpose, a 2-ary program-symbol `DEF` is defined. When a cell executes `DEF` it places its reading head on the right subtree. In order to be used, the left subtree must contains some labels of `NOP` program-symbols. The left subtree encodes a subnetwork whereas the right subtree corresponds to the main network.

Proposition 8 *Any cellular encoding scheme that includes the program symbols* `JMP`*,* `NOP_0`*,* `NOP_1` *and* `DEF` *is modular.*

In order to be able to connect an included subnetwork to the rest of the network in a functional way, another program-symbol of division called splitting division, and denoted `SPLIT` is required. A cell executes the `SPLIT` division as follows:

First, assume that the number of input links is equal to the number of output links. The cell splits into n children, where n is the number of links. The inheritance of the links is shown in fig. 3.8. The child cell with index i inherits the input link and the output link numbered i. Each child cell places its reading head on the same node of the grammar tree.

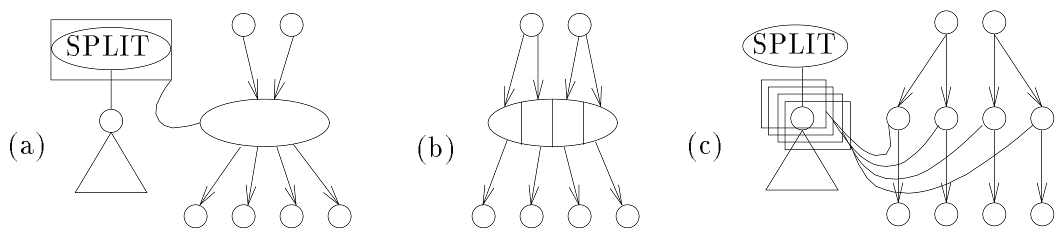

Figure 3.8: The splitting division. (a) before the division, (b) duplication of links in the input site, in order to make the total number of links equal to four, (c) creation of four neurons.

If now there are less input links than output links, some of the input links are duplicated in an homogeneous manner to make equal the number of input and output links, and apply the splitting. If the cell has less output links than input links, the output links are likewise duplicated.

The CE can be used to design a neural network that does a particular mapping. For example, in fig. 3.9 shows a neural network that simulates a Turing machine. The design method is to build networks of higher complexity by specifying how to connect together subnetworks of smaller complexity. The representation scheme uses this decomposition. Each subnetwork of neurons is named and represented separately. To represent a subnetwork h , instead of using the subtree t that encodes h, one represents a graph of cells which is obtained in the following manner. Take as an initial network graph, a reading cell whose reading head is placed on the root of t, and whose neighbors are a set of labeled cells. The labeled cells represents the neighbors of the cells that will include h, during the development of the complete neural network. Make this initial network graph develop, with two conventions.

- Whenever a cell reads a `JMP` program-symbol, freeze the development of the cell, introduce a new subnetwork g whose code is the subtree of the node labeled by `JMP`, and write the name of the included subnetwork, followed by the name of g in the circle representing the frozen cell.

- Whenever a cell reads a `SPLIT` program-symbol, freeze the development, introduce a new subnetwork g which code is the subtree of the node labeled by `SPLIT`, write `SPLIT` followed by the name of g in the circle representing the frozen cell.

If the subtree of the node labeled by `JMP` or `SPLIT` is only one node `END`, it is not necessary to introduce a new "virtual" subnetwork.

In the final network graph, the labeled cells indicate how the subnetwork h is to be connected to the rest of the network. Recall that the final network depends on the order of execution. In order to finish the representation, the order of execution of the frozen cells must be specified if needed. In the case of the Turing machine, it must be specified that the cells frozen on a `SPLIT` program-symbol have to execute this program-symbol only when all the neighbors are neurons. Thus, in the final encoding they must be preceded by a certain number of `WAIT` program-symbols so as to be correctly delayed. Note that this number is always a linear function of L, the initial value of the life register. These `WAIT` program-symbols can be encoded by means of a subnetwork which includes itself and does nothing but wait.

Fig. 3.9 (a) shows that the neural net is recurrent and has two layers: a processing layer and a mixing layer. The layers connect to each other in a one to one manner, thanks to two `SPLIT`

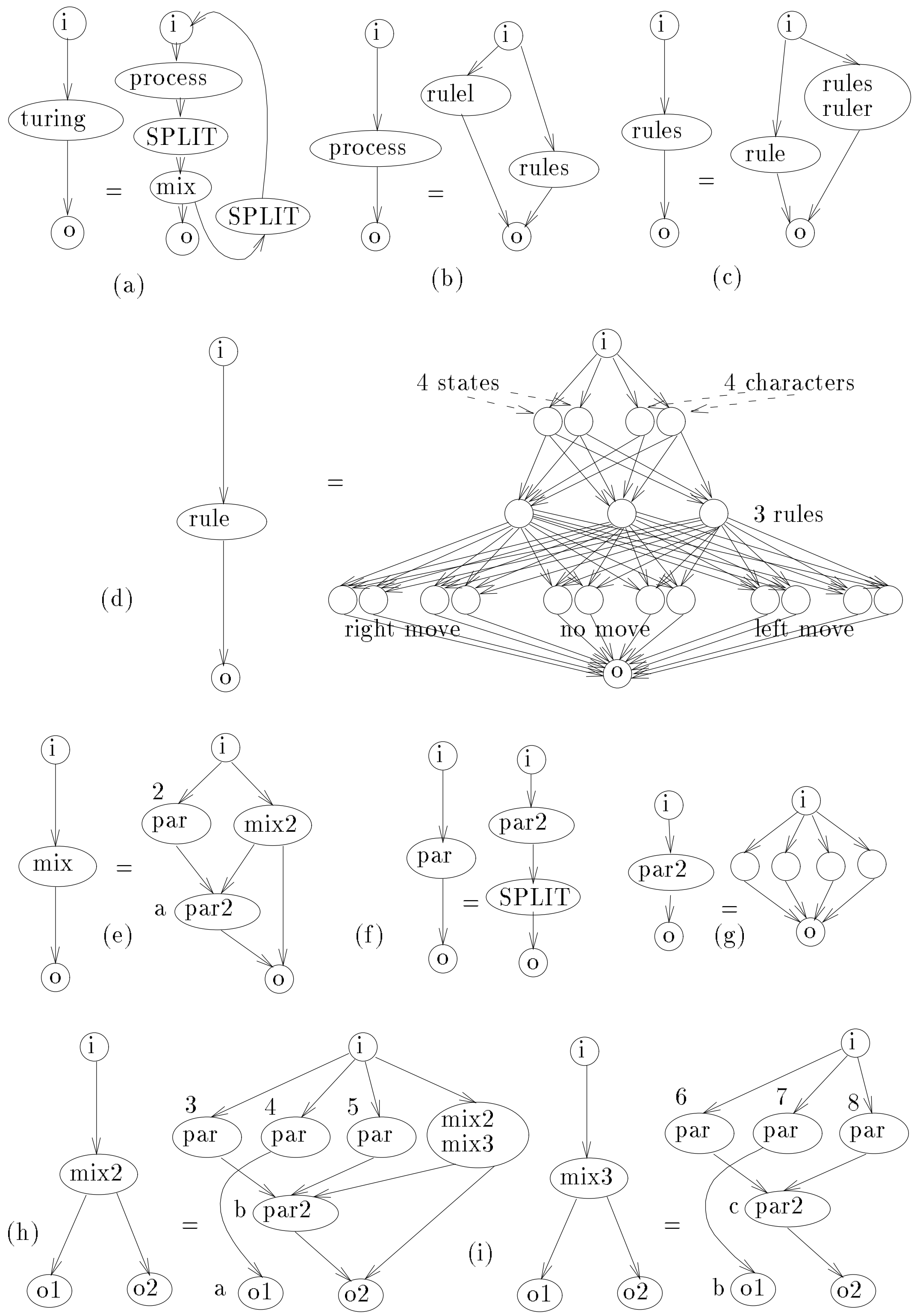

Figure 3.9: Encoding of a Turing machine.

inserted between the layers. Each time the data flows through the neural net, a transition of the Turing machine is simulated. Fig. 3.9 (b) and (c) shows that the processing layer applies in parallel L subnetworks of neurons called `rule` to a tape of L registers. Because of a slight difference, the leftmost `rule` is called `rulel`, and the rightmost, `ruler`. The subnetwork `rule` encodes all the transition rules of the Turing machine. Each group `rule` corresponds to a register of the Turing machine's tape. For each register, a character and a state are encoded. If the reading head is positioned on the register, the state encoded is the state of the Turing machine, otherwise the state is 0. On the input of the i^{th} `rule` is applied the binary code of the state and the character of the corresponding register. In the example, two bits are used for the state, and also two bits for the character. Hence if there are q states and an alphabet of c characters, the number of input units of `rule` is $i = \lceil \log_2(q) \rceil + \lceil \log_2(c) \rceil$. The number of output units of `rule` is $3 * i$, three rows of i cells. One row represents each of the possible three moves of the reading head of the Turing machine. The subnetworks `ruler` and `rulel` have only $2 * i$ units. Since they handle a register on the extremity of the tape, only two moves are possible. If the reading head of the Turing machine is not on a register, the state is 0, its code is the concatenation of $\lceil \log_2(q) \rceil$ 0, and the output of `rule` is 0 everywhere except for the $\lceil \log_2(c) \rceil$ units of the middle row whose activities are the same as the $\lceil \log_2(c) \rceil$ input units. This accounts for the fact that the character on the tape remains unchanged. If the reading head is on the register, it moves in one direction: left, stand by or right. The corresponding row of output neurons will indicate the next state, and the character to write. The 2 other rows will output zero values. Each hidden unit of `rule` implements one rule.

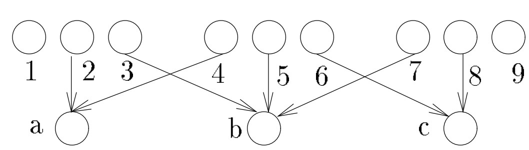

Figure 3.10: The action of the mixing layer.

The mixing layer must compute the next state and character of the tape registers, from the total output of the process layer. Therefore, $3i * L$ activities must be recombined into $i * L$ activities. Fig. 3.10 (a) shows how this must be done. Fig. 3.9 (e), (f), (g), (h) and (i) show how this is done. Each of the il activities can be computed with the logical `OR` applied to three of the $3il$ activities.

Now, let us define precisely what is meant by simulating a program on a machine that can have an arbitrary big memory with a parametrized neural net of finite size. First, define the possible functions implemented by a neural net as follows:

Definition 1 *A neural net N implements a function $f : \{0,1\}^n \to \{0,1\}^n$ if it has more than n input and p output units, and, if a given vector v of p bits is clamped to the first input units of N, all the other activities are initialized with 0, and N is relaxed with a synchronous dynamic, after stabilization, the activities of the first output units of N will be $f(v)$*

Now, consider a machine $M(L)$ with a memory of L bits and a program P that makes computations on bit strings. Run on some particular input bit strings, the program P will have enough memory, and will stop after some time, yielding a binary vector. By definition, these particular input bit strings form the input set of the function computed by P on $M(L)$, and the output of this function is given by P.

Definition 2 *A neural network $N(L)$ parametrized by the integer L, simulates a program P on a given type of machine $M(L)$ with a memory of L bits, if $N(L)$ implements the function computed by P on $M(L)$.*

Property 5 (power of expression) *A parametrized neural network encoding has a power of expression greater than a programming language if for every program P written with this language, there exists a code of a parametrized neural net that simulates the program P. Moreover, the size of the code is $O(s(P))$, where $s(P)$ is the number of bits needed to store the source code of P .*

In the case of the Turing machine, a program is just a list of transition rules. The preceding construction can produce a code of a parametrized neural network that simulates an arbitrary Turing machine program if the following adjustment is made: The input units of the processing layer that encode the state must not be connected to the input pointer cell. Their initial activity must be zero, except for the register of the tape which will initially contain the reading head of the Turing machine. Initially the state encoded in this register must be the initial state of the Turing machine q_0. Let us now evaluate the size of the codes. The number of bits needed to encode r transition rules is $s = (\log_2(q) + \log_2(c))r$. The size of the code of `par2` is $O(\log_2(q) + \log_2(c))$. The size of the code of respectively `rule`, `ruler`, `rulel` is $O(s)$. The size of the codes of the other subnetworks are constant. Hence the total size is $O(s)$.

Proposition 9 { *CYC*, `END`, `PAR`, `SEQ`, `SPLIT`, `CUT`, `INCLR` , `MRG`, `WAIT`, `VAL`, `NOP_0`, `NOP_1`, `JMP`, `DEF` } *has a power of expression greater than the "Turing machine language".*

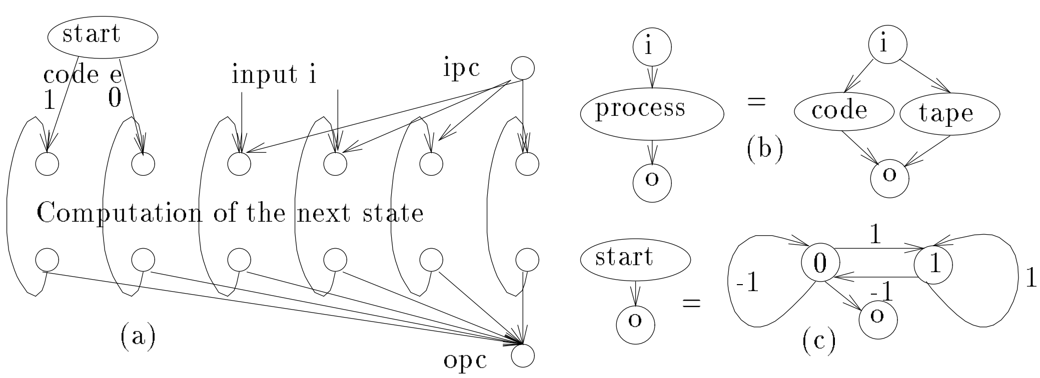

Figure 3.11: A neural network that simulates a Turing machine that simulates a neural network.(a) outline of the neural network, (b) Modification of the process layer, (c) the subnetwork `start`.

From the fact that CE has a power of expression greater than the Turing machine results in a sort of maximal property of compactness:

Proposition 10 { *CYC*, `END`, `PAR`, `SEQ`, `SPLIT`, `CUT`, `INCLR` , `MRG`, `WAIT`, `VAL`, `NOP_0`, `NOP_1`, `JMP`, `DEF` } *is functionally more compact than any other neural network encoding.*

In order to prove this, consider an arbitrary neural network encoding scheme E. Let e be the code of a particular neural network, encoded with this scheme. Call $s(e)$ the size of e. There exists a program with input e and i that builds the neural network encoded by e, runs the network on the input i, and returns the output of the network. This program can be executed by a Turing machine T. If the initial tape of T is the concatenation of e and i, the output of T will be the output of the neural network. Let N be the neural network that simulates T. Let $s(e)$ be the size of e. A string of $s(e)$ neurons must be added to N so that a string of $s(e)$ registers of the tape is automatically initialized with e. Fig. 3.11 (a) shows what the final neural network looks like. It depicts a neural network that simulates a Turing machine that develops a neural network encoded by $e = \overline{10}$, and simulates it. The simulated neural net has two input units. For simplification only the neurons corresponding to the characters of the tape are represented. The encoding of the final neural network must specify the string of $s(e)$ neurons, therefore, it is a bit more complex than the encoding in fig. 3.9. Fig. 3.11 (b) shows the first part of the code of the process layer. The subnetwork `code` will develop $s(e)$ subnetwork of the type `rule`. The i^{th} subnetwork is not connected to the input pointer cell. It is connected to the subnetwork `start` through an intermediate neuron with a weight 0 or 1, depending on the value of the i^{th} bit of e. The role of the subnetwork `start` is to send a one and then only zero values. The subnetwork `tape` develops a tape of L registers. The mixing layer must be slightly modified in order to mix $L + s(e)$ registers instead of only L. The size of the total encoding is a linear function of $s(e)$. The total size is therefore $O(s(e))$. A close analysis shows that the slope of the linear function does not depend on T ,and therefore, not on E either.

Another interesting property is scalability. Consider a problem of variable size, and a parametrized neural network encoding scheme. Suppose that the same code can be used to solve a problem of size one and problems of big sizes, using a different parameter. If the GA can find a code for a network that solves the problem of small size, it will not be harder for the GA to find a network for the problem of size 1.000.000, because the corresponding code is the same. The ability to address problems of big size is called scalability. The preceding remark allows us to define scalability as a theoretical property of the encoding scheme:

Property 6 (scalability) *Let (f_L) be a family of functions on a finite domain, where L is an integer index. An parameterized encoding is scalable with respect to (f_L) if there exists a code such that for any L the neural net encoded with a parameter L computes the function f_L.*

Consider the following definition:

Definition 3 *A family of functions f_n on a finite domain is said to be recursive if there exists a Turing machine T that computes $f_n(x)$ if n and x are initially written on its input tape. If f_n is recursive, $TM_{space}(f_n)$ is the minimal number of registers needed by T to compute every value of f_n.*

The following holds:

Proposition 11 *{ CYC,* `END`*,* `PAR`*,* `SEQ`*,* `SPLIT`*,* `CUT`*,* `INCLR` *,* `MRG`*,* `WAIT`*,* `VAL`*,* `NOP_0`*,* `NOP_1`*,* `JMP`*,* `DEF` *} is scalable with respect to the set of recursive family of functions with $TM_{space}(f_n)$ smaller than k^n for a certain k.*

In order to prove this proposition, we have to provide an encoding of an arbitrary Turing machine, such that the network developed with a size parameter L simulates the Turing machine with a finite tape of k^L registers, instead of only L registers as it has already been done. The encoding is similar to the one already proposed. As it is only a technical matter, it is not presented here. The family of languages that can be recognized by a Turing machine using less than k^n registers for a certain k is called PSPACE. PSPACE is so big that it includes almost all known languages [12]. Therefore, CE is scalable to all "every day" families of functions.

A cellular automaton can also be simulated with a construction similar to the case of the Turing machine. This time, the mixing layer must precede the processing layer, and its task is to reproduce at a location i, the concatenation of the states of the cell neighbors to the i^{th} cell.

Proposition 12 *{ CYC,* `END`*,* `PAR`*,* `SEQ`*,* `SPLIT`*,* `CUT`*,* `INCLR` *,* `MRG`*,* `WAIT`*,* `VAL`*,* `NOP_0`*,* `NOP_1`*,* `JMP`*,* `DEF` *} has a power of expression greater than cellular automata.*

Turing machine and cellular automata are universal computing devices. Cellular automata are particularly powerful. In general, a given algorithm can be efficiently programmed on a cellular automaton. It is possible to transform a set of transition rules of a cellular automaton into the CE code of a neural net in an efficient way. We can consider even more efficient computing device than cellular automata. The following property makes a step forward.

Property 7 (abstraction) *An encoding is abstract if it has a power of expression greater than a high level programming language.*

A high level programming language uses procedure and array data structure. Using an abstract encoding scheme allows us to reduce the problem of genetically finding a neural network to the problem of genetically finding a program that computes the target mapping. When there exists a small program that computes the target mapping, the space of program to search is smaller than the space of neural networks of bounded size, that contains some solution network. So an encoding that is abstract is likely to work better with the GA, if the mapping to learn can be computed with a short program. Since property of abstraction is hard to demonstrate, we will take the next chapter for that purpose. In fact, the proof is a compiler that transform a Pascal program into a CE which length is proportional to the length of the Pascal Program.

The last property is the following:

Property 8 *A coding scheme is grammatical if the object that is encoded can be translated into a grammar, a rewriting system.*

Proposition 13 *Cellular encoding is grammatical, it encodes a graph grammar*

The proof is in the appendix B. With a grammatical property, the genetic search of neural networks via cellular encoding appears as a grammatical inference process where the language is implicitly specified, not explicitly by positive and negative examples. This results in three new perspectives: First, ee will use a particular method to to search cellular encoding grammar (second part of this thesis). We will encode a grammar into a set of trees and evolve sets of trees with the genetic algorithm. This method can be used to infer other types of grammars. Conversely, results from the research in the grammatical inference field can be used to improve cellular encoding. Third, the study of cellular encoding as a grammar can be helpful to classify cellular encoding into the complexity classes that already exist among rewriting grammars. It seems that cellular encoding is a context sensitive, parallel graph grammar.

3.5 Conclusion

When applying Genetic Algorithm (GA) to neural network synthesis, it is now recognized that the key point is the scheme used to encode neural networks on the structure manipulated by the GA. Eight theoretic properties that rate the efficiency of a neural network encoding are proposed. A neural network encoding which has all the desired properties has been studied. This encoding, called Cellular Encoding (CE) uses an alphabet of 20 alleles. Each allele corresponds to a program-symbol that slightly modifies the topology and labels of a graph describing a network. A given combination of such program-symbols develops an initial graph of three cells (one ancestor cell plus two pointer neurons), into a neural network. The program-symbols are: `END`, `PAR`, `CLONE`, `SEQ`, `SPLIT`, `MRG`, `CUT`, `ON`, `OFF`, `INC`, `DEC`, `MULT2`, `DIV2`, `INCLR` , `MULT2LR`, `WAIT`, `NOP_0`, `NOP_1`, `JMP`, `DEF`. These program-symbols do not have integer arguments. This CE generates neurons with a single type of sigmoïd which is the sign function. It is easy to add other program- symbols if other types of sigmoïd are needed.

The properties of CE can be summerized as follows:

completeness Any neural network can be encoded, thus the GA can reach the solution neural network if it exists at all.

compactness The CE is topologically more compact than classical representations of neural networks, and functionally more compact than any other representation. This ensures that the codes manipulated by the GA are minimal in size. The shorter the codes are, the shorter the search space is, and the less effort the GA has to do.

closure The CE always develops meaningful architectures. It produces acyclic or recurrent architectures, depending on the initial graph. Both of these closure properties ensure that the chromosomes produced by the GA always give interesting neural networks. The GA does not throw away any codes. This increases efficiency.

modularity If a network can be decomposed into copies of subnetworks, the code of network is the concatenation of the of the subnetworks code, and a code that describes how to connect the subnetworks. This facilitates the formation of building blocks. The resulting regularities in the final architecture make it clear and interpretable. The code is more compact.

scalability A fixed size code encodes a family of neural networks. For any reasonable family of problems, there exists a code such that the i^{th} neural network solves the i^{th} problem. A parametrized problem of arbitrary size, say 1.000.000, can be as easy to solve as the same problem of size 2 or 3 because the code of the corresponding neural net is the same, only the size parameter changes.

power of expression The CE can be seen as a "neural network machine language". Its power of expression is greater than Turing machines and cellular automata. It is another proof of the extreme compactness of the coding.

abstraction It is possible to compile a Pascal program into the code of a neural network that simulates the computation of the program. The wide search space of neural network is transformed into a smaller search space of programs. It helps the GA to find neural networks for problems that can be solved with a short program.

grammars Cellular encoding encodes a graph grammar, this property is in fact the root of all the other properties.

Chapter 4

A neural Compiler

4.1 Introduction

An encoding scheme called "cellular encoding" has been described in the preceding chapter. Cellular encoding can be seen as a machine language for neural networks, able to describe a Turing machine, or a cellular automaton. One can use cellular encoding as a tool for designing neural networks that simulate a Turing machine or a cellular automaton. It is already known that a neural network can simulate a Turing machine using an infinite number of neurons. See [7]for variations on this theme, and the relation with cellular automata. In [29], Turing machine are simulated by neural networks having a finite number of neurons. In order to store an infinite amount of data, one uses the decimals of the neuron's activities. The activities must have an unbounded precision. These results have a theoretical interest rather than practical. It is indeed not usual and uneasy to program a Turing machine or a cellular automaton. In this chapter, we propose a method for compiling a program written in a high level language, towards a neural net. More than a proof that a program can be simulated by a neural net, the compilation is a practical method for building a neural network that solves a given problem. We will see that the compilation has other interests, like the design of huge neural networks, the automatic parallel compilation, the compilation of naive hybrid systems. Since the compiled networks have a finite number of neurons, and their activities have a finite precision, they have a finite memory. Hence, it is not possible to compile instruction for dynamic memory allocation. Nevertheless, recursive procedures can be compiled. During the run of the program, the depth of recursion must be smaller than a given constant. The performance of the compiler will be measured in term of the size of the code that is generated. The result confirms the efficiency of the compiler, and allows to demonstrate property of abstraction number 7, defined in the preceding chapter.

4.2 The stage of the compilation

The neural compiler has been programmed. The software is called JaNNeT (Just an Automatic Neural Network Translator). The aim of the software is solely to prove the validity of the compilation and to measure the performance of the product, like the number of neurons that are generated. JaNNeT encompasses three stages that are shown in figure 4.1. The input of the compiler is a program written in an enhanced Pascal. The output is a neural net that computes what is specified by the Pascal program. The Pascal is said to be "enhanced" because it proposes supplementary instructions compared to standard Pascal.

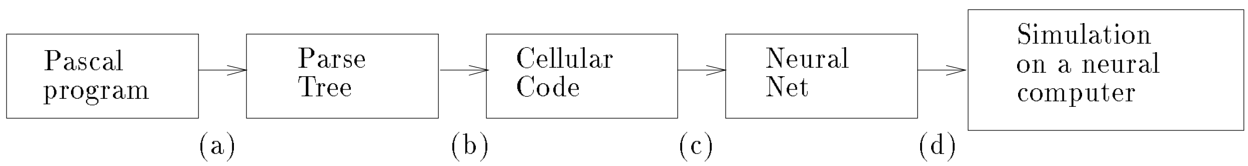

Figure 4.1: Stages of the neural compilation. (a) Parsing of the Pascal Program. (b) rewriting of the parse tree into a cellular code, using a tree grammar. (c) decoding of the cellular code into a neural network. (d) scheduling and mapping of the neural network on a physical machine. This stage is not yet done.

The first stage is the parsing of the program and the building of the parse tree. It is a standard technic that is used for the compilation of any language. Nevertheless, this parse tree has a somewhat unusual form. In appendix C.6, a tree grammar describes the kind of parse tree that we use. The third stage uses cellular encoding. It is simply the decoding of the cellular code that has been generated at the second step. This decoding uses the development of a network graph, seen in the preceding chapters.

The second stage is the heart of the compiler. This stage is a rewriting of trees. Rewriting means replacing a symbol by a group of symbols. In the simple case of word rewriting, we replace a letter by a sequence of letters. Starting from an initial word, one rewrites it into a new word, and then into another new word, and so on, until the final word. The rewriting of a tree (in a simple case) consists in replacing one node of the tree by a sub tree. The added subtree must specify where to glue the sub trees of the node that is being rewritten. Hence, the second stage consists in replacing each node of the parse tree into a sub tree labeled with program-symbols of the cellular encoding. When the program symbols of each sub tree will be executed by a cell c, they will make a local graph transformation, by replacing cell c into many other cells, connected to the neighbors of cell c. Each node of the parse tree can therefore be associated to a local transformation of a network graph of cells. Each node of the parse tree is corresponds to a *macro program symbol*, that does the local graph transformation. A macro program symbol makes a transformation bigger that the one done by a program symbol of the cellular encoding scheme. In most of the cases, a program symbol creates no more that one cell, or modifies no more than one register. It uses no more than a single integer parameter. A macro program symbol can create a lot of cells, and often needs more than one parameter. One needs many program symbols to encode a macro program symbol, this is why we use the name "macro program symbol". The presentation of the compilation method can be done without reference to the cellular encoding. One will consider the process at the level of macro program symbols. The macro program symbols can be implemented using sub trees of program symbols described in the appendix C.7.3. So this chapter can be read independently from the preceding ones.

In its present release, JaNNeT does not go until the end. It does not produce instructions that can be executed by a particular neural computer. It produces a neural network, which is a format that suits a neural computer with many processors dedicated to neuron simulation. Presently, the neural network generated by JaNNeT are simulated on a sequential machine. The fourth stage shown in figure 4.1 (d) consists in mapping the neural net on a physical architecture of a neural computer. This step must take into account things like the memory size of one neuron, the communications between processors, and the arithmetic precision of the computation.

4.3 The kind of neurons which are used

4.3.1 Sigmoid

A neuron is determined by the computation it does on its inputs. An ordinary neuron makes a sum of its inputs, weighted by the weights of the connections, and then, applies a function called sigmoid. We use different kind of sigmoid. Each one has a number associated to it. A cell has a register called `sigmo` in which is stored the number of the sigmoid of the future neuron. The four sigmoids that are used are listed in figure 4.2.

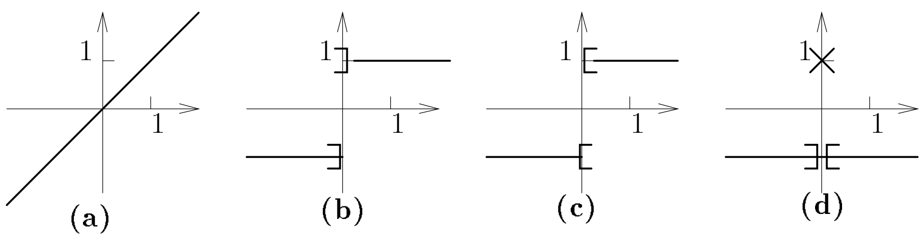

Figure 4.2: Different kind of sigmoids used. (a) is used to add two numbers. (b) and (c) are used to compare two numbers, (d) is used to test the equality of two numbers.

Moreover, we use non classical neurons that make the product of their inputs, and that divide the first input by the second input. This is used to multiply and to divide two numbers. It still is possible to simulate multiplication and division with classical neurons, but it would cost a lot of neurons.

4.3.2 Dynamic

A neuron computing an arithmetic operation with two operands, must wait for the two operands to come before starting to compute its activity. We use a particular dynamic. The neurons decide to start their computation using three rules:

- A neuron computes its activity as soon as it has received all the activities from its input neighbors.
- The input units are initialized with the input vector.
- A neuron sends its activity to the output neighbors, when it is initialized, of if it has just computed its activity.

We will not use any special threshold units to encode the threshold. At a given moment, all the neurons that can compute their activity, do it. This dynamic (denoted $\mathcal{D}_0$) is half way between the sequential dynamic and the parallel dynamic. There is no need for a global entity to control the flow of activity. Two other dynamics are used by the compiler. The dynamic $\mathcal{D}_1$ concerns neurons having exactly two inputs. As was the case for $\mathcal{D}_0$, the neuron waits until it has received an activity from both its neighbors. If the first input is strictly positive, the neuron send the activity of the second neighbor. Otherwise, the neuron do not propagate any activity. Dynamic $\mathcal{D}_1$ allows to block a flow of activities. The "EAND" neuron that we are going to see use this dynamic. There is a third dynamic called $\mathcal{D}_2$. Neurons with such a dynamic compute their activity whenever an activity is received from one of its input neighbors. Its activity is the activity of that particular neighbor. In order to work correctly, such a neuron must not received two activities at the same time. Dynamic $\mathcal{D}_2$ is used by the "AOR" Neuron. It allows to gather on the same neuron activities coming from different parts.

4.4 Microcoding

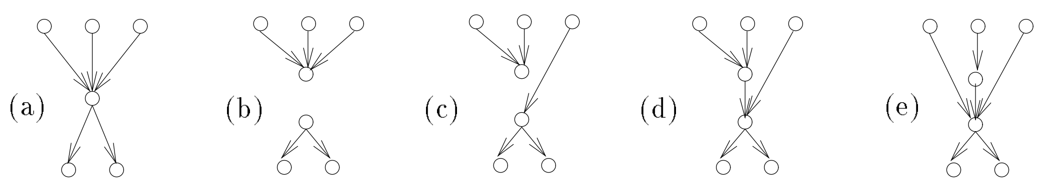

Figure 4.3: Microcoding of the program-symbol ADL 2. (Add on Link), 2 is an argument. This program symbol adds a cell on the link number 2. The microcode is DIVm<sm> 2 (a) before the division ADL, (b) division in two children. (c) the segment "m>" is analysed,. (d) the segment "s" is analysed (e) the segment "m<" is analysed.

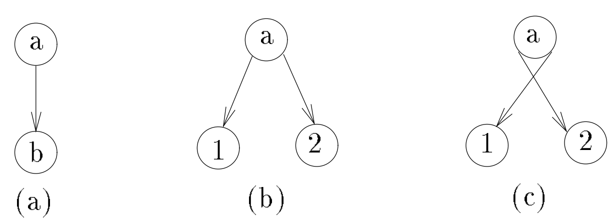

Figure 4.4: The difference between microcode with "d" and microcode with "r". (a) before the cell division; the cell which divides is labeled "b", it has an input neighbor labeled "a". (b) the input link is duplicated using a "d" microcoding. (c) the input link is duplicated using a "r" microcoding, the only change is the ordering of the links made by cell "a", between the first child labeled "1" and the second child labeled "2".

The preceding chapter was concerned with theoretical properties, we are interested here in a practical implementation. In order to make the compiler produce a more compact Cellular Code, we had to introduce many other division program-symbols, as well as program-symbols that modify cell registers, or that make local topological transformations, or that influence the order of execution. Each new program-symbol may have an integer argument that specifies a particular link or a particular register, this favors a compact representation. For implementation purpose, we have microcoded them. A microcode is composed of two parts. The first part specifies the category of the program symbol. It uses three capital letters. DIV indicates a division, TOP a modification of the topology, CEL, SIT, LNK a modification of respectively a cell register, a site register or a link register. A cell possesses an ordered set of input sub-sites for the links that fan in, and an ordered set of output sub-sites for the links that fan out. EXE, BRA indicates a program-symbol used to manage the order of execution, HOM refers to a division in more than 2 children, AFF denotes program symbols used only for the graph display.

The program-symbols are listed in appendix C.5. The second part of the microcode is an operand. For CEL, SIT, LNK the operand is the name of the register which value will be modified using the argument. If there is an arithmetic sign, the operator applies the arithmetic operation to the actual content of the register and the argument and places the result in the register. Else it simply sets the register with the argument. For EXE, the operand is the name of the particular order of execution. For BRA, the program symbol tests a condition on the registers of the cells or on the topology, if it is true, the reading head is placed on the left sub-tree, else it is placed on the right sub-tree. The operand is the name of the condition. For DIV, the operand is as list of segments. Each segment is composed of one letter and a possibly empty list of arithmetic signs. At the beginning, the cell divides, the first child cell inherits the input links, the second child cell inherits the output links. The segments are then analysed in the reversed order, they specify movements or duplications of links between the two child cells. Figure 4.3 gives an example of a DIV microcode analysis. If the letter is a capital, the links are duplicated or moved from the output site of the second child cell to the output site of the first child cell. If the letter is a small letter, the links are duplicated or moved from the input site of the first child cell to the input site of the second child cell.

- The letter 'm' or 'M' means: move a set of links,
- the letter 'd' or 'D' means: duplicate a set of links.
- the letter 'r' or 'R' is the same as 'd' or 'D' except that the added links are not ordered in the same way, by the neighboring cell. Figure 4.4 explains the difference between "d" and "r".
- The letter 's' means: connect the output site of the first child to the input site of the second child, it needs no arithmetic sign.

Arithmetic signs specify the set of links to be moved or duplicated. '<', '>', '=', specifies the links lower than, equal, or greater than the argument, '*' specifies all the links. '#' and '$' refer to the first and the last link, ' :' refers to links from the site of the neighboring cell, '−' is used when it is necessary to count the links in decreasing order, rather than in increasing order as usual, '−' placed before the three capital letter exchanges the role of the input site and the output site, '_' is used to replace the argument by the value of the link register. '|' is used to replace the argument by the number of input links divided by two.

A similar micro-coding is used for the operands of TOP, that locally modifies the topology, by cutting or reordering links. In this case, the single child cell inherits the output links of the mother cell, and the analysis of the segments indicates movements of links from the input site of the mother cell to the input site of the child cell.

In appendix C.10 there are also a number of program symbols which are used only for labeling cells. During the development of the network graph, the software that we have programmed can indicate the program symbols read by the cell. Neuron keep their reading head on the cellular code, on the last program symbol read. Therefore we use dummy program symbols to label leaves of the code. This enables us to see on the computer screen what function a given neuron performs.

4.5 New program symbols

In this section, we describe some program symbols used for the compilation of Pascal Programs. These program symbols are not described by their microcode, we explain them separately.

The program symbol `BLOC` allows to block the development of a cell. A cell that reads `BLOC` waits that all its input neighbors cells become finished neurons. This program symbol is often used in the compiler to avoid that a particular piece of cellular code is developed to early.

The compiler generates many trees of cellular code: one tree for the main program and one tree for each function. These trees are also called grammar trees, and have a number. The number is used by the `JUMP` program symbol. The program symbol `JUMP` without arguments has already been defined at section 3.4. Nevertheless, for the compiler, we find it more efficient to use a program symbol with an integer argument. The argument specifies the number of a grammar tree. A reading cell that executes the program symbol `JMP x` places its reading head on the root of the grammar tree which number is x.

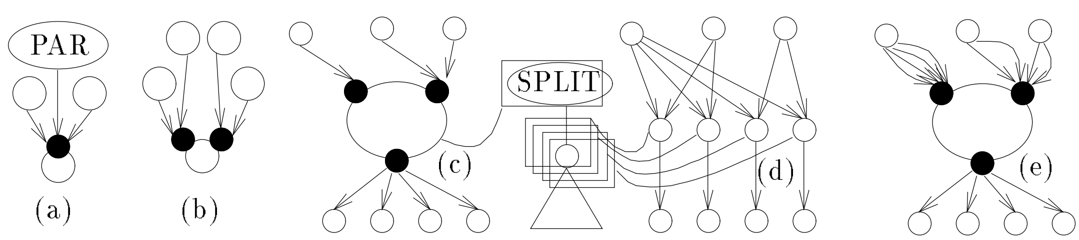

Figure 4.5: Propagation of the `begin-sub-site` flag. A sub-site is indicated by a small black disk. (a) Before the parallel division of the cell labeled `PAR` (b) After the parallel division, one see that a sub-site has been created by the neighbor, because the register `divisable` of the input site of that neighbor contains value 1. (c) Before the execution of program symbol `SPLIT`. (d) Result of the splitting division. (e) An intermediate stage.

We need to handle separately groups of link. We distribute the links into groups of links. Each group is contained in a sub-site. The sub-sites are created in a rather complicated way. The technical details that follow may be skipped without problems with the understanding of the rest of the chapter. The sub-sites are coded using a boolean flag `begin-sub-site` for each link. When the flag of a link is on, it indicates that a new sub-site begins at that particular link. The flag of the first link is always off. A link is encoded using two elements, one for the input site, and one for the output site. Each element is inserted into a linked list, as in figure B.3. The position of the element in the linked list determines the number of the link, in the input site and in the output site. Each element contains a different flag `begin-sub-site`. This is necessary, because a connection may start a new sub-site on its input site without starting any sub-site on the output site. Both the input and output site have a register called `divisable`.

When a cell division takes place, a link can be duplicated. The following rules determines the setting of the flag `begin-sub-site` of each elements of the connection which is duplicated.

Consider first the element that belong to the cell neighbor to the cell that divides. If the register `divisable` of the site corresponding to this element of the connection is 1, the value of the flag is duplicated. Otherwise, the value of the flag is copied in the element of smaller number, the other element has its flag initialized with the value 0. This rules means that if `divisable` is 0, there is no creation of sub-site. The old sub-sites are kept. If on the contrary, `divisable` is 1, a sub-site divides into two sub-sites. The figure 4.5 (a) and (b) gives an example of sub-site creation.

Second, we have to update the flag for the element of the connection that belongs to the cells created by the division (microcode beginning with `DIV`) or to a cell whose links have been modified (microcode beginning with `TOP`). First, we consider the case of a division. The value of the register `divisable` of the mother cell is duplicated. We have seen in the preceding chapter, that for `DIV`, the list of connections is created through the chaining of segments of connections. If the value of register `divisable` is 1, `begin-sub-site` marks are inserted so that one sub-site is created for each segment. Otherwise, no new mark are inserted, and the number of sub-sites is kept constant.

The distribution of links into group of links is intersting only with respect to the `SPLIT` program symbol. This program symbol has a particular behavior when the cell has many sub-sites. Some particular links are duplicated, so that the number of links on each sub-site is the same, whether it is an input or an output sub-site. Call n the number of links on each sub-site. The cell splits into n child cells. Each child cell has as much input links as there are input sub-sites in the mother cell. Similarly, each child cell has has much output links as there are output sub-sites in the mother cell. The value of all the registers are duplicated into the register of the child cell. If the register `divisable` is 1, each child cell has as many sub-sites as links, one sub-site for each link. If the register `divisable` is 0, no child cell has has any sub-sites.

4.6 Principles of the neural compiler

4.6.1 Structure of the compiled neural network

How can we do in order to simulate a Pascal program on a neural network? Figure 4.6 shows a simple example of compilation. Each variable V of the Pascal program contains a value. The variable is initialized at the beginning of the program. Then it changes following the assignments that are made. For each variable V, there corresponds as many neurons as there are changes in V's value. All these neurons represent the values taken by V during a run of the program. At a given step of the run, V has a given value v. The value v is contained in one of the neuron that represents V. This neuron is connected to the input neighbors that contain values of other variables, which have been used to compute v. The same neuron is connected with output neighbor that contain values of other variables. The input neighbors use the value of V to compute the new value of the variable that they represent.

The environment in compilation means the set of variables that can be accessed from a given point of the program. Before the execution of each instruction, the environment is represented by a row r of neurons. This row contains as many neurons as there are variables. Each of these neurons contains the value of the variable that it represents, at this particular moment.

An instruction is going to modify the environment, by modifying the value of some given variables. An instruction is compiled into a neural layer that computes the modified values, from the old values stored in the row r of neurons. The new values are stored in a row r_1 of neurons. Hence the whole Pascal program is compiled into a sequence of rows r, r_1, r_2, ... alternated with neural layers that compute the new values. There are as many such neural layers as there are Pascal instructions.

4.6.2 Compilation of a Pascal Instruction

The idea of the compilation method is to translate each word of the Pascal language into a modification of a network graph of neurons. At a coarse granularity, the compilation of a program is

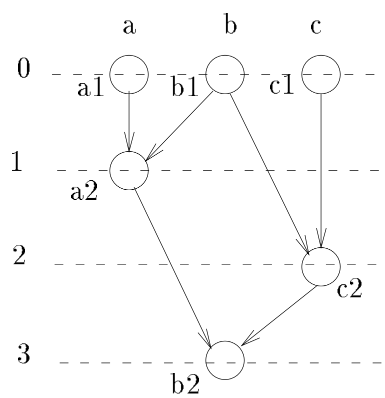

Figure 4.6: Result of the compilation of the Pascal program: *var a,b,c: integer; begin a=a+b; c=c+b; b=a+c; end.* Each dotted line corresponds to the compilation of one instruction. The neurons make a simple addition of their inputs (the weights are 1, the sigmoid is the identity). The ligne 0 represents the initial environment. It encompasses three values "a", "b" and "c" stored in three neurons a_1, b_1 and c_1. At line 1, the value of "a" has changed, it is now contained in a new neuron a_2, that computes the sum of "a" and "b". In order to make this sum, neuron a_2 is connected to neuron a_1 and b_1. The value of "c" changes at the next line, finally, the value of "b" also changes. One can see that the two first instructions: `a:= a+b; c:=c+b` can be executed in parallel by neurons a_2 and c_2. On this example, each variable is represented by two neurons during the run of the program.

decomposed in the compilation of each instruction. Each instruction is translated into a neural layer that is laid down, and a row of neurons that contains the environment modified by the instruction. So the compilation is a construction process. The compiler builds the row of neurons that contains the initial environment, by translating the part of the parse tree where the variables are declared. Then the compiler lays down consecutively neural layers, by compiling the program instruction by instruction.

This idea of progressive building of the compiled neural network can be applied with a granularity smaller than a Pascal instruction. A Pascal instruction can be decomposed into words of the Pascal language, these words are organized according to a tree data structure called parse tree (see figure 4.7). we can associate each word of the Pascal language with a local modification of a network graph of cells, so that the combined effect of these small modifications transforms a single cell into a neural layer that computes what is specified by the Pascal instruction. This is done using a system similar to the cellular encoding. The neural network will be developed during many steps. Certain neurons will have a copy of the parse tree, and a reading head that reads a particular node of the parse tree. They will be called reading cell rather than neuron, because their role is not to do the computation corresponding to a neuron, but to divide, so as to create the neural layer associated to the Pascal instruction. Each cell reads the parse tree at a different position. The labels of the parse tree represent instructions for cell development that locally act on the cell, or on connections of the cell. We will see that these instructions can be decomposed into a sequence of program symbols of the cellular encoding scheme. Therefore, they will be called "macro program symbols". During a step of development, a cell executes the macro program symbol read by its reading head, and moves its reading head towards the leaves of the parse tree.

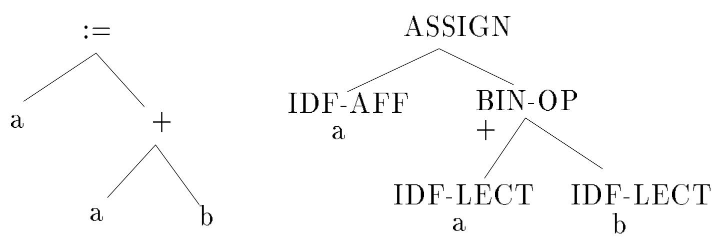

Figure 4.7: Parse tree of instruction "*a:=a+b*". On the link side: the parse tree is labeled with words of the Pascal program. On the right, the labels are made of two parts: the first part specifies the kind of the node, the second part is an attribute that stores the function of the node. For example,`IDF-AFF` indicates that the node contains the name of a variable that is modified, `IDF-LEC`: the name of a variable which is read, `BIN-OP` the name of a binary operator like the addition, `ASSIGN` indicates that we are doing an assignment, the left sub tree must contain the name of a variable to modify, the right sub tree contains the expression of the new value. For nodes `IDF-AFF` and `IDF-LEC`, the attribute contains the name of the variable to modify or to read. For `BIN-OP`, the attribute contains the name of the particular binary operator. The `ASSIGN` node does not have particular attributes.

A cell also manages a set of internal registers. Some of them are used during the development, while others determine the weights and the thresholds of the final neurons. Consider the problem of the development of a neural net for compiling the Pascal instruction "`a:=a+b`". Although the development of this instruction causes the creation of four cells, at the end of the development, only one cell is left.

The development begins with a single cell called the ancestor cell, connected to neurons that contain the initial values of the environment. Consider the network described on the right of figure 4.8. At the beginning, the reading head of the ancestor cell is placed on the root of the parse tree of the Pascal instruction, as indicated by the arrow connecting the two. Its registers are initialized with default values. This cell will divide many times, by executing the macro-program symbols associated to the parse tree. It will give birth to the neural network associated to the parse tree. At the end, all the cells loose their reading head, and become finished neurons. When there are only finished neurons, we have obtained the desired neural network. The following example describes the compilation of a Pascal instruction. It is important to keep in mind that in the figures, the input and output connections are ordered. The position of the connection on the circle that represents the cell, codes the connection number.

4.6.3 Compilation of the Pascal program

At the preceding sub-section, we have shown how the parse tree of a Pascal instruction can be interpreted as a tree labeled with macro program symbols. When these program symbols are executed by cells, they develop a neural layer that computes what is specified by the Pascal instruction. This method can be generalized to the whole Pascal program. The total program can be represented by its parse tree. Figure 4.13 represents the parse tree of the Pascal program "`program p; var a : integer; begin read(a); write (a); end.`". The first nodes of the parse tree

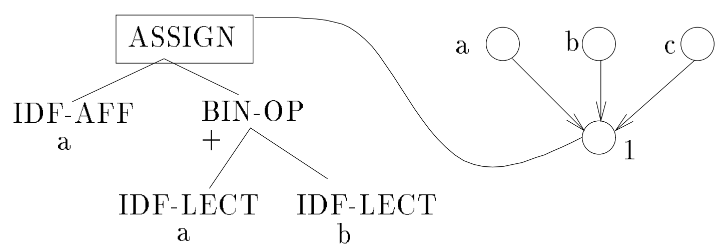

Figure 4.8: Development of the neural network corresponding to the Pascal instruction "`a:=a+b`", on the right side of this figure. The development begins with an ancestor cell labeled "1" connected to three neurons labeled "a", "b" and "c". On the left side, we represent the parse tree of the Pascal instruction. The arrow between the ancestor cell and the symbol `ASSIGN` of the parse tree represents the reading head of the ancestor cell. The input and output connections are ordered. It is the position of the arrow on the circle of the cell that codes the the connection number. For example, the link to "b" has number 2.

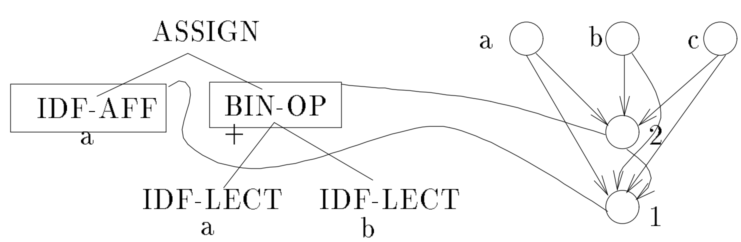

Figure 4.9: Step 1, execution of the macro program symbol `ASSIGN`, read by the ancestor cell "1" of the preceding figure. Cell "1" divides into two child cells "1" and "2". Both child cells inherit the input links of the mother cell. Child "2" is connected to child "1". The reading head of cell "1" now points to the node `IDF-AFF` and the one of cell "2" points to the node `BIN-OP`

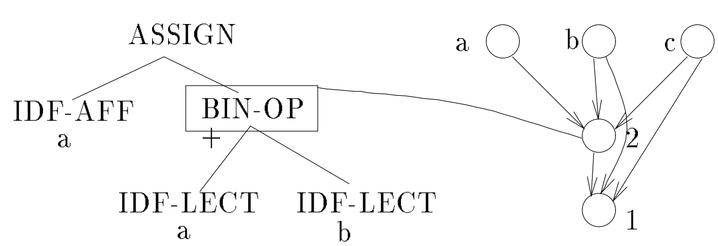

Figure 4.10: Step 2, execution of the macro program symbol `IDF-AFF`, read by the cell "1". Cell "1" deletes its first link, and moves its fourth link back in the first position. It looses its reading head and becomes a neuron.

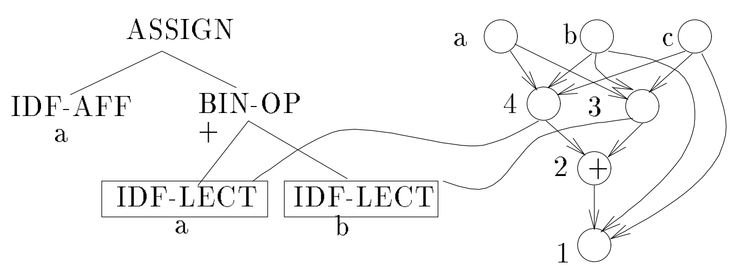

Figure 4.11: Step 3, execution of the macro program symbol `BINOP` read by cell "2". Cell "2" divides into three cells "2", "3" and "4". Cells "3" and "4" inherit the input links of their mother cell, and are connected on the output to cell "2". Whereas cell "2" inherits the output links of the mother cell, looses its reading head and becomes a neuron. The sign + inside the circle indicates that neuron "2" makes the addition of its inputs.

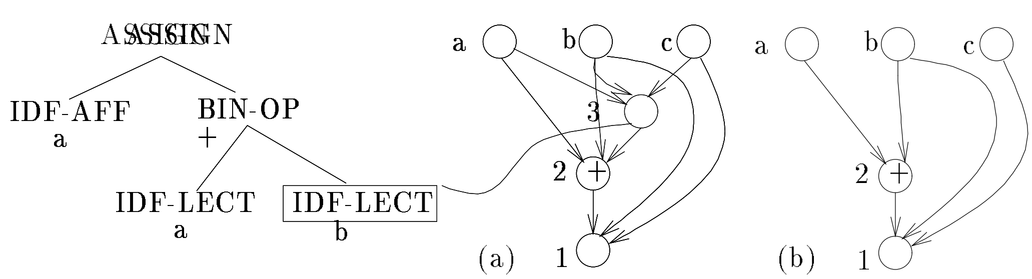

Figure 4.12: (a) Step 4: Execution of the macro program symbol `IDF-LEC`, read by cell "4". The cell "4" disappears, after having connected its first input neighbor to its output neighbor. (b) Step 5: Execution of the macro program symbol `IDF-LEC`, read by cell "3". The cell "3" disappears, after having connected its second input neighbor to its output neighbor. There are no more reading cells, The compilation of the Pascal instruction "`a:=a+b`" is finished. The ancestor cell "1" is connected to the three neurons that contain the values of the modified environment. It can start to develop the next instruction.

are used to declare variables. They will create the first layer of neurons that contain the default values of the variables. The following nodes of the parse tree correspond to the instructions. They will create many layers of neurons that make the computations associated to these instructions.

Consider the starting neural network, on the right half of figure 4.14. The input pointer cell and the output pointer cell to which the ancestor cell is linked, do not execute any macro program symbols. At the end of the development, the input pointer cell points to the input units of the network, and the output pointer cell points to the output units. The development of the network is reported in appendix C.1. The graph of cells reaches a size of 6 cells, at the end only two neurons are left.

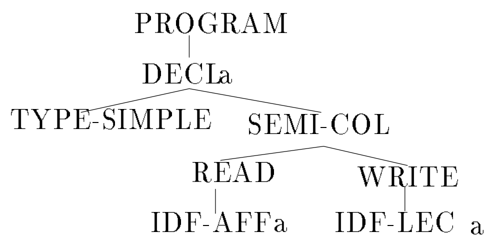

Figure 4.13: The parse tree of the Pascal Program: "`program p; var a :  integer; begin read(a); write (a); end.`" This parse tree does not have the usual form. It contains nodes for the declaration of variables, and the labels are not organized in a standard way. This particular form helps the implementation of the neural compiler. The appendix contains a BNF grammar that describes the form of these parse trees.

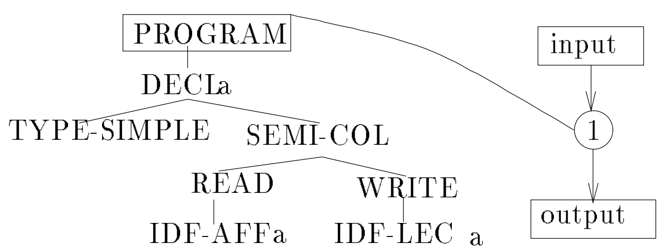

Figure 4.14: Development of the neural network that simulates the Pascal program: "`program p; var a :  integer; begin write (a); end.`", on the left half of this figure. The development begins with an ancestor cell labeled "1" and represented as a circle. The ancestor cell is connected to an input pointer cell (box labeled "input") and an output pointer cell (box labeled "output"). On the left, we represent the parse tree of the Pascal program. The arrow between the ancestor cell and the parse tree represents the reading head of the ancestor cell.

4.7 The macro program symbols

We have seen at the preceding section, the method of neural compilation and, as an illustrative example, the compilation of a small Pascal program. The method is based on the association of each label of the parse tree with a macro program symbol. We call each macro program symbol with the name of the associated label. A macro program symbol m replaces the cell c that executes it, by a graph d of reading cells and neurons. This graph is connected to the the neighbors of c. The term "neighbor" refers to two cells connected either with a direct connection or, indirectly, through special neurons called pointer neurons. The graph d must specify where the reading cells are going to read in the parse tree. In fact the reading cell will read most of the time a child node of the node labeled by m. Since each node is read by exactly one cell, the number of reading cells of d is equal to the number of child nodes. Nevertheless, it can happen that certain reading cells are blocked, and finish their development later. In this case, the graph d contains more reading cells as there are child nodes.

The parse tree that we use are described by a tree grammar in appendix C.6. This tree grammar details the list of macro program symbols and how they are combined. In the following of this chapter, we will refer to the non terminal of this grammar. We classify two kinds of macro program symbols, the macro program symbols of the type "expression", they correspond to labels generated using the non terminal <expr>, and produce neurons that are used to compute values. And the macro program symbols of the type "modification of environment" that are used to manage the environment.

In this section, we are going to present the macro program symbols that have not been illustrated in the exemple shown in the preceding section. We will put forward an invariant pattern which is the rewriting context. When a cell is being rewritten by a macro program symbol, its neighbor cells are always of the same type, in the same number. If the environment contains n variables, the cell will have $n+2$ input links, and two output links. The first input (resp. output) link points to the input (resp. output) pointer cell. The second input link is connected to a cell called "start cell", the last n input links point to neurons that contain the values of the variables. The second output links points to a cell labeled "next". The "next" cell's role is to connect the graph generated by the macro program symbol, with the rest of the network graph. If the macro program symbol is of the type "expression", the "next" cell is a neuron, else it is a reading cell.

The first macro program symbol executed during the compilation of a program is the macro program symbol `PROGRAM` (figure C.1). This macro program symbol creates the invariant pattern. By recurrence, this invariant is kept afterwards because the reading cells generated by each macro program symbol have their neighbors that verify the invariant pattern.

How can we associate a macro program symbol to each label of the parse tree? 1) We check the invariant pattern, 2) we try to minimize the number of neurons that are created and 3) we make sure that the macro program symbol does the desired job. In the case of a macro program symbol of the type expression, the generated sub-network must compute the right value, in the case of a macro program symbol of the type "modification of the environment", we check that the modification is correct.

From time to time we use cellular encoding in our description. In the practical implementation, the macro program symbols are decomposed in program symbols. For each macro program symbol, these decompositions are reported in the appendix. The program symbols of cellular encoding implement small modifications of graph. For example, they never create more than two cells (except the `SPLIT`) and they have a single parameter. In the contrary, the macro program symbols create a lot of cells, connected in a complex manner. The decomposition of macro program symbols into program symbols is an efficient way of implementing macro program symbols. It is a simple, and quick way. It allows to summerize all the macro program symbols in two pages of expression with parenthesis, in the appendix C.7 and C.8. During the compilation, it often happens that some cells, after the execution of a macro program symbol, block their development until all their input neighbors are neurons. When they unblock, they go to execute a piece of cellular encoding in order to do a certain graph transformation.

4.7.1 Kernel of the Pascal

We consider in this sub-section, programs that are written with a reduced instruction set: the kernel of the Pascal, that allows to write only very simple programs. We consider for the moment only scalar variables, we will see arrays later. The macro program symbols `DECL` are in a string. The effect of each `DECL` macro program symbols is to add an input link to the ancestor cell. This link is connected to a cell that is itself connected to the "start" cell with a weight 0 (figure C.2 and C.3). This last connection ensures that the neurons corresponding to the variable will be activated, and will start the network. We suppose that each variable is initialized with the value 0.

The sequential execution is implemented using a string of reading cells, associated to a list of instructions. The cell of the string at position i reads the sub parse tree of instruction number i of the list. On figure C.4, the input neighbors of cell "1" are neurons that contain the values of the environment. The cell at position $i+1$ has a single input neighbor which is the cell number i. The cell number 2 must delay its development until cell number 1 has finished to develop the sub-network that modifies the environment according to instruction 1. The cell 2 blocks its development until all the input neighbors cell have became neurons.

The assignment is the most simple example of instruction. In order to understand figure 4.9, recall that each variable corresponds to an input connection number. The storage of a value in a variable is done by connecting the neuron that contains the value to the ancestor cell. The connection must have the right number, in order to store the value in the right variable. The management of the connection number is done by the macro program symbol `IDF-AFF`, figure 4.10. Similarly, in order to read the content of a variable, it is enough to know its link number and to connect the neuron that contains the value of the variable to the neuron that needs this value.

The input units of the neural net correspond to the values read during the execution of the Pascal program. If in the Pascal program the instruction `read a` appears, there will be one input unit created. By setting the initial activity of this input unit to v, the variable a will be initialized with v. The output units correspond to the values written during the execution of the Pascal program. If the instruction `write a` appears, there will be one output unit created. Each time the instruction `write a` is executed, the value of this output unit will be the value of a. A neuron is an input unit if it is connected to the input pointer cell. It is an output unit if it is connected to the output pointer cell. The macro program symbol `READ` adds a connection from the input pointer cell to the neuron that represents the variable, and the macro program symbol `WRITE` adds a connection between the neuron that represents the variable and the output pointer cell (figure C.5 and C.8).

The binary arithmetic operator (figure 4.11) are translated in a tree of neurons, each neuron implements a particular arithmetic operation using a particular sigmoid. Unary operators are implemented in the same way, only two cells are created instead of three. The first cell will develop the sub network corresponding to the sub expression to which the unary operator is applied. The second one will place its reading head on the cellular code that corresponds to the unary operator. This cellular code describes how to develop a sub-network that computes the unary operator.

Until now, the macro program symbols were rewriting one cell into less than 3 cells. We are going to present a new kind of macro program symbols. The `IF` rewrites a cell into a graph of cells whose size is proportional to the size of the environment. This represents a step in the complexity of the rewriting. This new class of macro program symbols can be implemented using sub-sites and the `SPLIT` program symbol. The Pascal line `if a then b else c` can be translated by the boolean formula: `(a AND b) OR ( (NOT a) AND c)` or by the neuronal formula: `(a EAND b) AOR ( (NOT a) EAND c)`. The `TRUE` value is coded on 1, and the `FALSE` is coded on -1. A logical `NOT` can be realised with a single neuron that has a -1 weight. On figure C.12, we can see a layer of 6 neurons `EAND`, divided into two sub layers of three neurons. These sub layers allow to direct the flow of data in the environment towards either the sub-network that computes the body of the `THEN` or the sub-network that computes the body of the `ELSE`. Recall that the neuron `EAND` (extended `AND`) has a special dynamic. It is an `AND` for integers, whose two inputs are a boolean and an integer. If the boolean is `FALSE`, the neuron `EAND` outputs nothing. If the boolean is `TRUE`, it outputs the integer. After that, a neuron `AOR` arithmetic `OR`, retrieves either the output environment of the `THEN` sub-network, or the output environment from the `ELSE` subnetwork, and transmits it to the instruction that follows the “if then else”. The `AOR` also have a special dynamic. They update their activity as soon as one of their input neighbor has changed its activity. The dynamics of `AOR` and `EAND` are described at section 4.3.2.

When a constant is needed in a computation, this constant is stored in the bias of a neuron n whose sigmoid is the identity. This neuron is linked to the "start" cell that will control its activity, that is to say, at what time, and how many times, the neuron n sends the constant to other neurons that need it.

The instruction `while <condition> do <body>` corresponds to a recurrent neural network in which an infinite loop of computation can take place using a finite memory. No memory allocation is done inside the body of a loop. The flow of activity enters the loop through a layer of `AOR` neurons. This layer is a buffer. Then the flow of activity goes through a network that compute the condition of the while. Following the value of the condition, the flow of activities goes out of the loop or is sent back in the body of the loop. In the last case, it goes again through the buffer of neuron `AOR`, after having passed through a layer of neurons which act as a synchronizer. (figure C.16). The synchronization is done to avoid that a given neuron in the body of the loop updates its activity two times, whereas in the mean time, another neuron has not updated it at all. In the case of two encapsulated loops, the neurons corresponding to the inner loop come back to their initial state when the computations of the inner loop are finished. They are ready to start another loop if the outer loop commands it. The neuron "start" is useful especially in the body of a loop. It is considered as as local variable, and therefore, there is a copy of it in the buffer of neurons `AOR`. It is connected to all the neurons that contain constants used in the body of the loop, and activate these neurons at each loop iteration. This is essential for the neural network computations.

The compilation of the `REPEAT` is similar to the `WHILE`. The computation of the condition is done at the end, instead of at the beginning.

4.7.2 Procedures and functions

We are going to explain here how to compile procedures and functions. We will use an example of Pascal Program in order to illustrate how to pass parameters, how to call, and how to return from a procedure or a function. Here is the program:

```
program proc_func

Var glob: integer               (* global variable*)

Procedure proc2(paf2: integer)  (*formal parameter *)
var loc2: integer;              (*variable local to  procedure proc2*)
   begin
   ...
   end;

Procedure proc1(paf1: integer)  (*formal parameter *)
var loc1: integer;              (*variable local to  procedure proc1*)
   begin
   proc2(pef2);                (*parameter  *)
   end;                        (*passed  by value to proc2*)

BEGIN                          (*body of the main program*)
   proc1(pef1)                 (*parameter *)
END.                           (*passed  by value to proc1*)
```

We will use a slightly modified invariant pattern, where the environment contains three variables. The first variable is global, the second is a parameter passed to a procedure, the third is one is a local variable of a procedure.

In all the figures, the first output link of cell "1" points to the output pointer cell and the second link points to "next" cell. If, sometimes, the inverse is represented, it is only for a purpose of clarity in the representation. It allows to have a planar graph.

Let us imagine that the context in which we are, at a certain point of the parse tree, is the moment where `proc1` is calling `proc2`. The environment on the neuronal side encompasses the global variable, that is, the input pointer cell, the "start" cell, plus the Pascal variable `glob`. The environment also contains the variables that are local to `proc1`, `pef1` and `loc1`. We want to translate the calling of procedure `proc2`. The first thing we do is to suppress the local variables of `proc1`, from the environment. This is done by the macro program symbol `CALL-P` described figure C.17.

Then we develop the parameters passed by value from `proc1` to `proc2` (macro program symbol `COMMA` figure C.19). Finally, the ancestor cell will develop the body of `proc2`. The body of `proc2` begins with the macro program symbol `PROCEDURE` that inserts a cell. When the body of the procedure `proc2` will be translated, this cell will pop the local variables of `proc2` (cellular code `POP` figure C.22). After that, a cell (inserted by a `CALL-P`) executes a cellular code `RESTORE` that allows to recover the local variables of `proc1`. It is not obligatory to put aside (using cut and restore) the local variables of `proc1`. The interest of it, is that each variable can be assigned a fixed connection number.

The macro program symbols `CALL-P`, `CALL-F` and `POP` use two parameters: The parameter `GLOBAL` is the number of global variables in the environment. The parameter `LOCAL` is the number of local variables. The macro program symbol `CALL-F` is simpler than `CALL-P`. It does not need to create a cell that will restore the environment of the calling procedure, because a function does not return an environment, but simply a value. The macro program symbol `FUNCTION` has no effect. It is not necessary to create a cell that will pop the environment of the function. The macro-program symbol `RETURN` also has a null effect.

4.7.3 The arrays

The arrays are a very important data structure in a programming language like Pascal. We had to use a special kind of neurons in order to be able to handle arrays. These neurons are called pointer neurons. They are used only to record the array data structure in a neural tree. The pointer neurons are nodes of the neural tree. The data are the leaves of the neural tree. The use of pointer neurons ensures that the ancestor cell possesses exactly one input link per variable. It allows to handle the variables separately, when one want to read or to modify them. Using pointer neurons, one can represent array with many dimensions. The invariant must be extended. It now contains pointer neurons. The figure C.25 uses an example of extended invariant, that represents an array 1D with two elements. Figure C.26 uses a neural tree of depth 2 to represent an array with 2 dimensions. A tree of depth d can represent an array with d dimensions. The use of many layers of pointer neurons for storing an array with many dimensions is necessary if one want to be able to handle each column separately, in each dimension.

When an array variable is declared, the structure of neural tree is created for the first time. For this purpose, the type of a variable is coded in the parse tree as a string of macro program symbols `TYPE-ARRAY` (see figure C.27). Each of these macro program symbol is associated to one dimension of the array, and creates one layer of pointer neurons, in the neural tree. Each macro program symbol `TYPE-ARRAY` has a single parameter which is the number of columns along the dimension corresponding to that macro program symbol. The string of `TYPE-ARRAY` is ended by a macro program symbol `TYPE-SIMPLE` which was already described in figure C.3.

Many macro program symbols must handle neural trees. The must make a recurrent exploration of the neural trees. For example, the `IF, WHILE, X-AOR, X-SYNC` are macro program symbol that must re-build the neurons containing the current environment.

Since we do not know the depth of the tree, the only way is to process the tree recursively. It is not a problem, since the cellular encoding allows recursive development. We use a program symbol `BPN x` that tests whether the neighbor which number is x id a pointer neuron, or not. Depending on the result of the test, a different cellular code will be executed. In one case a recursive call is done, in the other case, a terminal processing is done.

The reading of an array, like `a:=t[i1, i2]` is translated by an unusual parse tree. The parse tree used for `t[i1, i2]` is indicated figure C.29. The reading of an array is interpreted as the result of a function `1lect-index` written in cellular code that has $d+1$ inputs for an array of d dimensions. The inputs are the d indices plus the name of the array. The function `1lect-index` returns the value read in the array.

The writing of an array like for example `t[i1, i2]:=v` is translated in a parse tree of the same kind as in the case of the reading. It is indicated figure C.30. The writing of an array of d dimensions is interpreted as the result of a function `1affect-index` written in cellular code, with $d+2$ inputs and one output that is an array. The inputs are the d indices, the value to assign, the name of the array. The function`1affect-index` returns the modified array after the writing. One can see on the figures that the neural networks developed, either for reading or writing, are big. The number of neurons is proportional to the number of elements in the array. But, if the indices used to read or to write the array can be known at compilation time, the complexity of the developed neural net can be reduced. The added neural net has a fixed size that does not depend on the number of elements in the array.

4.7.4 The enhanced Pascal

In this sub-section, we describe the compilation of new instruction that have been added to the standard Pascal. These instructions exploit the interest of a neural compiler. One has added the instruction `CALLGEN`, that allows to call a function defined by its cellular code. At the beginning of the Pascal program, there must be an instruction `#include <file-name>` that indicates the name of a file where the cellular codes of the functions to be called are stored. The syntax of the `CALLGEN` is the following: `<idf0> := CALLGEN("code-name", <idf1>, .., <idfk>)`. the result of the call will be a value assigned to the variable `idf0`. The code name indicates the particular cellular code of the called function. `idf1, ..  idfk` are the parameter passed to the function. The macro program symbol `CALLGEN` has the same effect as `CALL-F` except that the body of the called function is not specified by a parse tree, but by a cellular code that has been defined before the compilation takes place. The `CALLGEN` includes a neural layer that computes the called function. The neurons that contain the values of `idf1, ..., idf0` are connected to the input of the included neural network. The output of the included neural network is connected to the neurons representing the variable `idf0`. Before executing the body of the `CALLGEN`, all the neurons that contain the value of the environment are suppressed. The philosophy of the `CALLGEN` is to define a number of functions that can be used in various context, or to include some neural building blocks, that have been trained. The neural networks developed using a `CALLGEN` never need global variables. Hence it is simpler to suppress them before the calling, in a systematic way. Exemple of predefined functions in the appendix C.9 are:

- function `gauche` returns the left part of an array
- function `droite` returns the right part of an array
- function `concat` allows to merge two arrays.
- function `int-to-array` adds a dimension to an array
- function `array-to-int` suppresses a dimension to an array
- function `random` initialize an array with random values in $\{-1, 0, 1\}$

One of the main goal of the compiler is automatic parallelization of a program written with a divide and conquer strategy. In this strategy, a problem of size n is divided into two sub problems of size $n/2$ and then into 4 sub-problems of size $n/4$ and so on until problem of size 1. One uses the array function `gauche` and `droite` in order to divide the data of an array into two equal part. One share the columns in the first dimension into two sets of columns of equal size. If the number of columns is odd, one of the sub array has one column more than the other.

We need to stop the division process when the size of the problem is 1. One must test during the development of the neural net, when the number of columns of an array is 1. Depending on the result of the test, a different part of the parse tree must be developed. In one case, the part of the parse tree corresponding to the decomposition must be developed. In the other case, the part of the parse tree corresponding to the problem of size one must be developed. We use a static `IF`. The syntax is the same as the normal `IF`, `IF THEN, ELSE` are replaced by `#IF #THEN, #ELSE`. Nevertheless the condition of the `#IF` is always the same expression. This expression is written `t=1`, where t is an array. The macro program symbols `#IF` and `#THEN` test whether the array t has one or more than one column, in the first dimension. If it is the case, the ancestor cell goes to read the left sub-tree of the `#ELSE`, if not, it goes to read the right sub tree.

4.8 Results of the compilation

In this section, we propose a few examples of compilation that illustrate the principles of the compiler and its interest. In order to run a compiled network, on must initialize the input unit with the value read during the execution of the Pascal program. The neural net makes its computation with a dynamic which is halfway between parallel and sequential. When the network is stable, the computation is finished. The output units contain the values that are written by the Pascal program.

4.8.1 Compilation of a standard Pascal program

Figure 4.8.1 shows a network compiled with the following program:

```
program p;
type tab = array [0..7] of integer;
var t : tab; i, j, max, aux : integer;
begin
  read (t);
  while i < 7 do
  begin
```

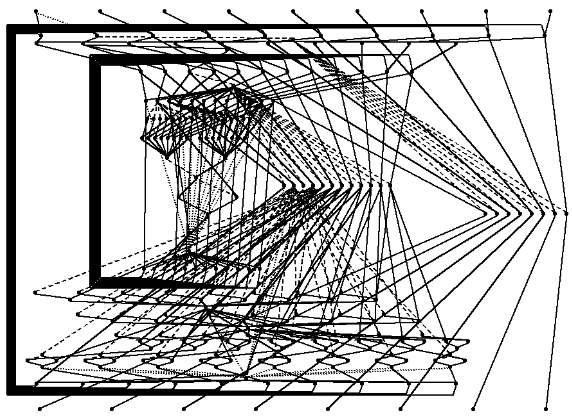

Figure 4.15: A neural net for sorting 8 numbers

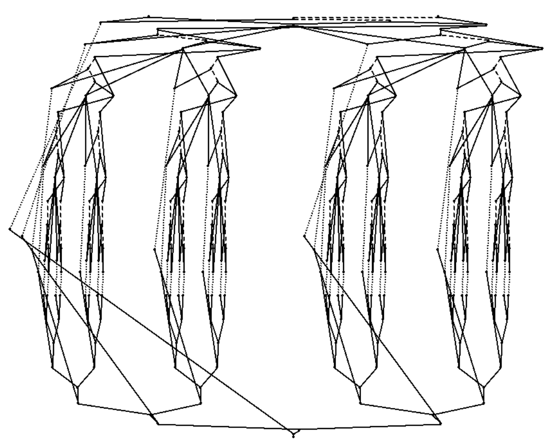

Figure 4.16: A neural net for computing Fibonacci number, developed until depth 6.

```
    j := i + 1;max := i;
    while j <= 7 do
    begin
      if t[max] < t[j] then max := j  fi;
      j := j + 1;
    end;
    aux := t[i];t[i] := t[max];
    t[max] := aux; i := i + 1;
  end;
  write (t);
end.
```

There are 9 input units, the "start" cell plus the 8 values to be sorted. There are 8 output unit that correspond to the instruction `write(t)`. Recurrent connections are drawn using three segments. Clearly one can see the two encapsulated loops that are mapped on two recursive sets of connections. Figure 4.16 shows a neural net that has been compiled with the following program:

```
program Fibonacci;
```

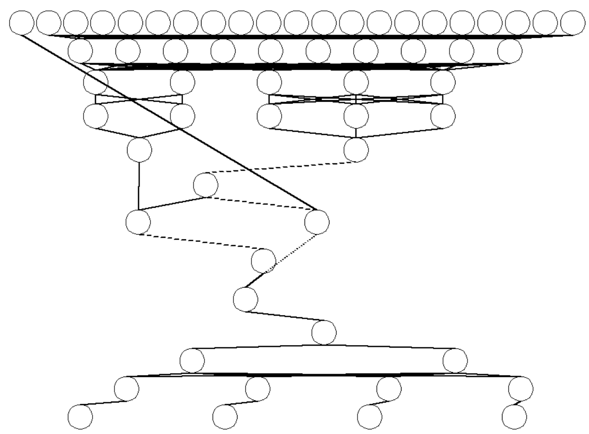

Figure 4.17: An hybrid neural network that simulates the behavior of an animal

```
var resul: integer; a : integer;
function fib(n: integer):integer;
begin
  if (n<=1) then resul := 1
  else resul := fib(n-1)+fib(n-2) fi;
  return (resul);
end;
begin
  read (a); write (fib (a));
end.
```

This example illustrates the fact that if the program to be compiled uses recursive calls, the depth of recursion must be bounded, before the compilation begins. Here the depth of recursion has been bounded by 6. Hence this net can compute `fib(i) i<6`. In fact the number 6 is the value used to initialize the life register of the ancestor cell.

4.8.2 How to use the `CALLGEN` instruction.

We have added to the Pascal language two instructions: `CALLGEN` and `#include` that allow to include in the final network, smaller networks whose weights have been found using a learning algorithm. The code of these nets are stored in a file which name is indicated by the instruction `#include`. We now show an example of Pascal program that uses the `CALLGEN` instruction. The compiled net is shown in figure 4.17.

```
#include "animal.a"

program animal;
type tab=ARRAY[0..19] of integer; tab2=ARRAY[0..9] of integer;
var pixel: tab; feature: tab2; position, is-bad, move: integer;

function opposite(position: integer):integer;
begin  return(1-position); end;

begin
```

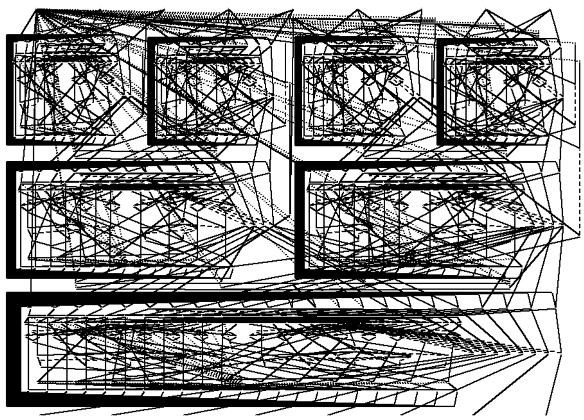

Figure 4.18: A neural net for the parallel sorting of 8 integers.

```
  read(pixel); feature:=CALLGEN("retine",pixel);
  position:=CALLGEN("position_object",feature);
  is-bad:=CALLGEN("predator",feature);
  if(is-bad=1) then move:=opposite(position)
  else move:=0 fi;
  write(CALLGEN("motor",move));
end.
```

In the file `animal.a` are stored the cellular codes of different layered neural networks. The network `retine` inputs some pixels, and outputs a list of relevant features. The network `position-object` determines from these features, the position of an object, if an object lies in the visual field. The neural net `predator` determines whether the object is a predator animal. The neural net `motor` commands the muscles of the legs. The whole compiled neural net simulates the behavior of an animal in front of a predator. The input units are the pixels, and the output units are the neurons of the leg muscles. This neural net can be stored in a library, in order to be included in a bigger neural network. This example illustrates how easy it is to interface the Pascal language with predefined neural nets. The compiled neural net encompasses many neural layers, plus one part that is purely symbolic. This is the part that corresponds to the instruction `If`. So the compiler not only links together various neural networks, but it allows to make symbolic computation from the outputs of the neural building blocks, and it can manage the input/output.

4.8.3 How to use the #IF instruction

We have added to the Pascal language an instruction `#IF #THEN #ELSE` that allows to test a condition at compile time. The tested condition always has the same form. One test whether the first column of an array has more than one column. This instruction allows automatic parallelization of a program written with a "divide and conquer" strategy. In this strategy, a problem of size n is decomposed into two sub problems of size $n/2$ and then four problems of size $n/4$. We use the instruction `#IF` with some other predefined functions, that enrich the array data structure. for exemple `gauche` and `droite` extract the left part and the right part of an array, `int-to-array` and `array-to-int` add or suppress a dimension. We now report an example of Pascal program that uses the instruction `#IF` The compiled net is shown figure 4.18

```
#include "array.a"
program merge sort;
const infini=10000; type tab = ARRAY [0..7] of integer; var w : tab;

function merge(t: tab; n: integer; u: tab; m: integer): tab;
var v: tab; i, j, a, b: integer;
begin
  v:=CALLGEN("concat", t, u); a:=t[0]; b:=u[0];
  while(i+j<n+m) do
  begin
    if(a<b) then
    begin  v[i+j]:=a;i:=i+1; if (i<n) then a:=t[i] else a:=infini  fi; end
    else
     begin v[i+j]:=b;j:=j+1; if (j<m) then b:=u[j] else  b:=infini fi; end
    fi;
  end;
  return(v);
end;

function sort(t: tab; n: integer) : tab;
begin  #IF t=1
       #THEN return(t)
       #ELSE return(merge(sort(CALLGEN("gauche",t),n DIV 2),  n DIV 2,
                 sort(CALLGEN("droite",t),n-(n DIV 2)), n-(n DIV 2)))
       #FI;
end;

begin read(w); write(sort(w,8)); end.
```

Function fusion is written in standard Pascal. Function `sort` uses the enhanced Pascal in order to implement the "divide and conquer" strategy. The file `array.a` contains the library of the cellular codes that allow to compute the function manipulating arrays. The compiled neural net has 9 inputs: the start cells plus 8 integers to sort. It has 8 outputs that show the sorted list of the 8 integers. The structure of the net can be understood. The first layer of recursive neural nets outputs 4 sorted lists of two integers each. The second layer outputs two sorted lists of four integers. Although recursive, the function `sort` can be compiled on a finite graph, because it is applied to a finite array of integers. Since we have used the divide and conquer strategy, the network sorts n integers in a time which is $O(n)$ (if the neuron can do their computations in parallel). We have also compiled many other programs using the divide and conquer strategy. The convolution, the matrix product, the maximum of an integer list, are programs that are compiled on a neural net able to do the job in a time $O(\ln(n))$.

4.9 Application of the compiler

JaNNeT is a process based on a new paradigm: Automatic building of neural network using an algorithmic description of the problem to be solved. This paradigm is at the exact opposite of the existing trend which present neural networks as machine that learns by themselves.

We are now going to show that JaNNeT can reproduce and even improve three kinds of compilation. JaNNeT is therefore a process able to cumulate the interest of these three compilations.

4.9.1 Neural network design

The interest of a language for describing neural networks like SESAME([19]) is to facilitate the design of large and complex neural networks. These languages propose to define some "building block" neural networks. and then to put them together in order to build more complex neural networks. The latter can again be composed, and so on... This hierarchical design is very practical. The instruction `CALLGEN` allows to achieve the same effect with JaNNeT. When it compiles a neural net, JaNNeT produces its cellular code. This cellular code can be stored in a file, and called using CALLGEN, so as to be included in a bigger neural network.

This technic allows a modular compilation, from the compilation point of view. It permit a hierarchical design, from the neural network design point of view. But JaNNeT goes further than the language of description like SESAME. JaNNeT gives the possibility to combine neural building blocks without mentioning the physical connections between the network. The specification is made on a logical, soft level. A human being understand much better a soft description where it is enough to describe the logical step of a computation, rather than an intricated set of connections. JaNNeT allows the design of huge neural networks. In the near future, machines with millions of neurons will be available, and the advantages of a logical description will become obvious. Moreover, the JaNNeT compiler produces a graphical representation of the neural net that takes into account the regularities of the graph. To our knowledge, this is the very first software that can do that.

4.9.2 Tool for the design of hybrid systems

JaNNeT is a process that can compile a Pascal program towards a neural network. If we would like to compile a base of rules used in expert systems (artificial intelligence), towards a neural network, we just need to write the base of rules in Pascal, as well as the inference motor. Hence JaNNeT includes the compiler of base of rules, used in the so called hybrid systems, which goal is to combine artificial intelligence and neural networks. Nevertheless, JaNNeT can be used for the design of hybrid system in a different way. The compiler can put in the same model of neural networks, algorithmic knowledge, (that knowledge is compiled) and "fuzzy knowledge", (that one is learned in a fixed size neural networks). The compiler can build naive hybrid systems that encompass two layers. The first layer is learned and it is sub symbolic. It computes symbols by grounding them on fuzzy and distributed data. The second layer is compiled, it makes a computation on these symbols. The resulting hybrid system is said to be naive because the symbols are not linked together.

4.9.3 Automatic parallelization

The numerical examples shows that JaNNeT can automatically parallelize problems like sorting or matrix-vector product, matrix-matrix product, vector-vector product. JaNNeT can parallelize algorithms written with a "divide and conquer" strategy. These kind of algorithms divide a problem into two sub-problems of half size. Each of these sub-problems is again divided into two sub-sub-problems. etc... until a problem of size one is reached, that can be solved in a constant time. Since all the problems of size one are independent, they can be solved at the same time. There lies the potential for parallelism.

In its actual version, JaNNeT cannot be considered as an automatic parallelizer, because it does not produce instructions for a particular parallel machine. It produces a neural network, which is a representation adapted to a parallel computer, because a neural network is an intrinsic parallel model. The final step consists in mapping this network on the architecture of a parallel machine. This step must take into account the size of the memory of each processor, the communications between processors, and the arithmetic precision of the computation.

One cannot yet claim that JaNNeT is a full automatic parallelizer. Nevertheless, the neural networks produced by JaNNeT have a 3D shape. One of the dimension of this structure represents the time. It is possible to project the network along this dimension on a 2D array of processors, so as to map the neural net on a 2D array of processors.

JaNNeT is also not a "full" parallelizer, because the program must be written using a divide and conquer strategy. But most of the algorithm can be programmed in that way, and the advantage compared to the traditional approach described in [26] is that the user does not need to add in his program indications about where (on which processor) to store the data.

Part II: Use of Cellular Encoding with the Genetic Algorithm

In the second part of this thesis, we will study neural network synthesis using the genetic algorithm (GA) and cellular encoding (CE). We suppose that the reader knows cellular encoding. The first chapter of this second part describes simulation done with the GA alone, the use of a learning will be introduced in the next chapters.

Chapter 5

Genetic synthesis of Neural Networks without learning

5.1 Genetic Programming

In the introduction of this thesis, we have presented the GA as an optimization procedure that manipulates codes called chromosomes. We presented the basic case where the chromosomes used are fixed length bit strings. It is possible to use any kind of structures for the chromosomes, provided that two operators of mutation and recombination are defined on the structures. Recombination creates a new offspring chromosome by mixing information coming from two parent chromosomes. Mutation slightly modifies a chromosome. Recombination allows an exchange of information between chromosomes, mutation performs a kind of random hill-climbing. The actual trend in the GA community is to put as much as possible information in the structure of the chromosome and in the operators, to ensures that the chromosomes generated by the GA are always valid, i.e. their syntax is correct and they can be evaluated, The structure of the chromosome should contain the syntactical information, so that the GA does not have to bother about syntax. Koza has shown that the tree data structure is an efficient encoding which can be effectively recombined by genetic operators. *Genetic Programming* has been defined by Koza as a way of evolving LISP computer programs with a genetic algorithm [16]. In Koza's Genetic Programming paradigm the individuals in the population are LISP S-expressions which can be depicted graphically as rooted, point-labeled trees with ordered branches. Since our chromosomes (grammar-trees) have exactly the same structure, the same approach is used for generating and recombing grammar trees. The set of alleles is the set of cell program-symbols that label the tree, the search space is the hyperspace of all possible labeled trees. During crossover, a sub-tree is cut from one parent tree and replaces a subtree from the other parent tree; the result is an offspring tree. The subtrees which are exchanged during recombination are randomly selected. Figure 5.1 gives an example of cross-over.

We used a Genetic Programming algorithm but with some differences from Koza. We do not stop the GA after a given number of generations. Instead we stop when elapsed time is greater than a variable T. We use a mutation rate $t_m = 0.005$, each allele has a probability t_m to be mutated into another allele of the same arity. We found that mutation consistently improves the Genetic search. Koza considers the depth of the individuals in the initial population. We take into account only the number of nodes and generate an initial population of chromosomes having exactly the same number of nodes l_0. When the exchange of sub-tree is done during cross-over, the size of the tree is not allowed to increase above l_{max}. We have designed a sequential genetic algorithm that uses local selection and mating based on a geographical lay out of individuals. Individuals

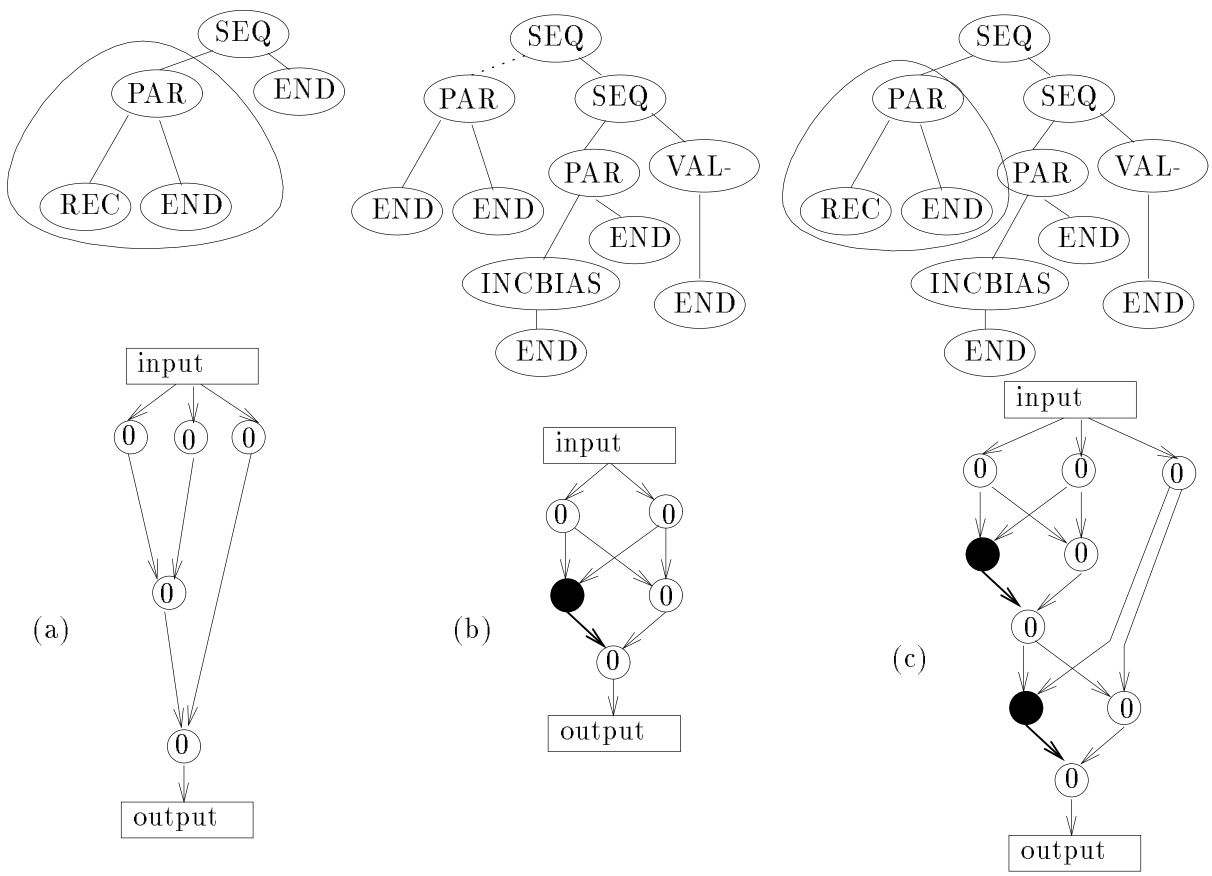

Figure 5.1: An illustration of recombination between two trees. The second order neural network is developed from the resulting offspring, where the life parameter of the ancestor cell is initialized to 2. The father tree encode an OR neural net. The selected subtree to be cut is encircled. The mother tree encodes a XOR neural net. The dotted line indicates the subtree which is to be pruned and replaced. The resulting offspring encodes a general solution for the parity problem.

reside on a 2-D grid and mate with the best chromosome found during a random walk in the neighborhood. This implements isolation by distance [4], such that different local regions on the grid may display local trends in terms of evolutionary development. Individuals are deleted by using an approximation of the scheme of GENITOR [31]. We select randomly 50 individuals among the population, and delete the worse individual in this sample, instead of deleting the worse individual in the whole population. The population size p and the number of generations g are not fixed. Rather, the GA deduces p and g using a function that yields the expected number of generations given a population size ($g = p/10$ in our simulation) and the total allocated CPU time. The GA is a steady state model which uses the following strategy to decide when to create a new individual and when to delete an individual from the current population. The GA evaluates the average time taken for the evaluation of one chromosome. It then computes the number of generations that could still be processed, keeping the current population size p. If it is higher than $p/10$ then it creates an individual, else it deletes an individual. Using this strategy, when the allotted computation time has elapsed the final population size p and number of generation g will be such that $g = p/10$.

Each time a new individual is created and evaluated with a time t the average evaluation time is updated using $t_{avg} = (p * t_{avg} + t)/(p+1)$. Since the average evaluation time increases along the generations, the number of individuals in the population decreases. Thus, the population size can dynamically adapt to the change in the amount of time required for fitness evaluation. In the early generations, the chromosomes are quickly evaluated and a large number of solutions can be explored. In the last generations, the evaluation can become, as we will see, very expensive, and only a few chromosomes stay in competition.

5.2 The fitness

A very important task in genetic neural networks is the fitness used to evaluate a neural network. This section describes the fitness used to evaluate a neural network.

Let us define f_{tar} as the boolean function to be computed by the neural network, and f_{nn} the boolean function which is actually computed by the neural network. The GA produces tree-shaped grammars which always develop a valid phenotype, that is, a connected graph of neurons. Now, the input and output units must be specified. Fig. 2.1 shows the initial graph. The input and output pointer cells are pointers that define the set of input and output cells. At the end of the rewriting steps, the ordered output links of the input pointer cell define an order in the input units. Similarly, the ordered input links of the output pointer cell define an order in the output units. So the number of input and output units is also encoded in the chromosome. The advantage is that the code is self consistent, it can be decoded without knowing what is the dimension of the input and the output space of the target mapping. Because of self-consistency, the code can be use as a building-block, and concatenated in another code to include the corresponding neural net, for a purpose of modularity. If the number of input or output units is not that which is expected, the boolean function f_{nn} which is computed by the neural network can still be defined as follows: if there are too many visible units, be they input or output units, the supernumerary neurons are deleted in decreasing order; if there are too few, the lower part of the input vector is entered and the upper undefined components of the output vector are set to 0.

The following task is to define a measure of how close f_{nn} is from the target function f_{tar}. Instead of the usual mean squared error, a metric based on information theory is used. The reason why the fitness is based on information is the following. An information based measure is smoother in relation with the natural topology on labeled trees defined as follow: the neighbors of a given tree are found by removing adding, or changing a node. Let us give a concrete short example of the kind of smoothness induced by information theory. Consider a tree t that develops a neural net which computes the complement of f_{tar}. Suppose that the output space is $\{0,1\}$. In order to get the right mapping, one has to add a single neuron that does a logical NOT. Adding a branch to the tree t, it is possible to build a tree t' that adds this neuron. With respect to the preceding topology t and t' are near. A fitness based on mean squared error would give the lowest mark to t and the highest to t'. Although there is a lot of useful information in t, t would have no hope to reproduce, and all this information would be lost. A fitness based on information gives the same mark to t and t'.

Here is a short definition of the information and the mutual information. Consider a finite set E with a probability density d and two random variables X, Y in E, with probability density pd and qd, The information of X is:

$$I(X) = \int_E \varphi(p(x))d(x); \varphi(u) = u * \log_2(u)$$

The mutual information between X and Y is:

$$H(X,Y) = \int_{E\times E} \varphi(\frac{p\otimes q(x,y)}{p(x)q(y)})p(x)d(x)q(y)d(y)$$

The following holds:

$$0 \leq H(X,Y) \leq I(Y); H(X,Y) = I(Y) \Longleftrightarrow \exists g : E \rightarrow E; p\otimes q(x,y) \neq 0 \Rightarrow y = g(x)$$

We now show how to use information theory in order to define a fitness. First, let us assume that the output space is {0,1}. Consider a random vector V in the input space, with a uniform probability density. The fitness measure is defined by:

$$\gamma(f_{nn}, f_{tar}) = \frac{H(f_{nn}(V), f_{tar}(V))}{I(f_{tar}(V))}$$

$H(f_{nn}(V), f_{tar}(V))$ can be seen as the information brought by f_{nn} on f_{tar}. We divide it by $I(f_{tar}(V))$ in order to get a fitness between 0 and 1. A mathematical analysis shows that when the fitness reaches the value of 1 the neural net produces the right output (if not it produces the complement).

If now the output space is $\{0,1\}^p$ we apply the preceding approach to each of of the p components and make the average of the p resulting fitnesses.

Finally let us consider a family of functions which are assumed to share the same type of algorithm. We call f^L_{tar} a member of this family, where L indexes the size of the problem and is stored as the initial value of the *life* register. For each L, $L = 1,\ldots,L_{max}$ a neural net N_L develops and computes an approximation f^L_{tar} of the target f^L_{nn}. The total fitness is:

$$\Gamma(L_{max}) = \sum_{L=1}^{L_{max}} \gamma(f^L_{nn}, f^L_{tar}) \tag{5.1}$$

Experiments show that individuals are very bad during the first generations. The neural network N_1 has a poor fitness and the neural networks N_L for $L > 1$ have a fitness 0. Hence, it is not necessary to develop the neural networks N_L for $L > 1$ in the early generations. The following rule decides exactly when it is necessary to develop the neural network N_2, and then N_3 and so on, until $N_{L_{max}}$.

> **Developing Rule:** If N_1, N_2, ..., N_L have already been developed, then, continue to develop N_{L+1} if N_L's fitness is greater than a given threshold r. If $L > L_{max}$ stop the development.

The threshold r is a parameter to fix, according to the particular application. With this rule, in the early generations, only one or two networks are developed. Whereas in the last generations, up to L_{max} neural networks are developed, thus time is saved. Since the size of N_L is also more than L times bigger than the size of N_1, for $L = 1,2,\ldots,L_{max}$, the evaluation of the fitness can last $L_{max} * (L_{max}+1)/2$ times longer. We used $L_{max} = 50$ for the parity, thus, the ratio can be up to 2500. The developing rule has another interest: Since each term in the sum lies between 0 and 1, the total sum lies between 0 and L_{max}. With the developing rule, only the first terms from the L_{max} terms are summed. If the threshold r is near 1, The developing rule says that the algorithm continue to develop the following network only if the current one does the right mapping. Finally, if the fitness lies between L_0 and L_0+1, it means that the L_0 first networks do the right mapping, and the L_0+1^{th} one is not yet correct. So the fitness can be clearly interpreted to tell how far the problem as been solved.

5.3 Simulation with the parity

Objective:	Find the CE of a neural network family that computes the parity of an arbitrary large size
Terminal set:	`SETBIAS 0` `SETBIAS 1`
Function set:	`PAR SEQ WAIT VAL- REC`
Fitness cases:	Random patterns of $L+1$ bit to input in the neural net N_L. the number of fitness cases is determined by the GA
Raw fitness of one Neural Net	The mutual information between the function computed by the neural net and the target function
Standardized fitness of one Neural net	raw fitness divided by the information of the target function It ranges between 0 and 1
Standardized fitness	The sum of standardized fitness of the developed neural net. It ranges between 0 and 50
Parameters:	$T = 1800s, t_m = 0.005, l_0 = 20, l_{max} = 30$
Success Predicate:	The standardized fitness equals 50. That is the 50 first networks are developed and their fitness is maximum

Table 5.1: Tableau for genetic search of a neural network family for the parity problem up to 51 inputs

Table 5.1 summarizes the key features of the problem for evolving a Celllar Encoding that generates a family of Neural network for computing the parity function of arbitrary large size. The GA tries to find a chromosome that develops a family of neural networks (N_L), such that N_L computes the parity function of $L+1$ inputs. The parity of L binary inputs is the sum of all the inputs, modulo 2. The threshold defined in the developing rule: $r = 0.7$. We develop the $L_{max} = 50$ first neural networks. So the solutions are tested for the parity problem with up to 51 inputs. Nevertheless, the GA generates solutions to the parity of arbitrary large size, because the recurrence is learned. The program-symbol `PAR SEQ WAIT VAL- REC` have been described in chapter 2. Since no program-symbols `INCLR` or `DECLR` are used, the link register is not modified and its value remains the default value which is 1. Therefore the program-symbol `VAL-` sets the weight of the first input link to the value "-1". The program-symbol `SETBIAS 0` and `SETBIAS 1` set the threshold of the neuron to respectively 0 and 1. We used a highly reduced set of program-symbols in this first experiment. We wanted to make the GA search as easy as possible.

The population size p and the number of generations g are not specified. Instead, a variable T specifies the time we allocate to the GA. The GA deduces p and g out of a function that yields the expected number of generations given a population size ($g = p/10$ in our simulation) and the total allocated CPU time T. The GA computes p and g by evaluating the average time taken for the evaluation of a chromosome. This average changes along the generations, and therefore the GA must update its deductions permanently. Since the evaluation time increases, the number of individuals decreases (from 1100 to 550 in fig. 5.2). Thus the population size can dynamically adapt to the change in the time of fitness evaluation. In the early generations, the chromosomes are quickly evaluated, and a large number of solutions can be explored, so as to select many different peaks in the fitness landscape. In the last generations, the evaluation is very heavy (up to 2500 times longer, as already noticed), and only a few people on each selected peak stay in competition.

The number of patterns that is necessary to evaluate the fitness with a good enough precision is not specified either. Instead, the GA determines this number by comparing the accuracy of the actual fitness with the variance of the population. Fig. 5.3 shows that the better the individuals, the more patterns are allocated.

Since the GA turns out to produce huge networks, we introduce another parameter to save time: If the ratio of the number of cells of the network graph with respect to the number of visible units exceeded a threshold of 15, the development was stopped, and the fitness of the neural net

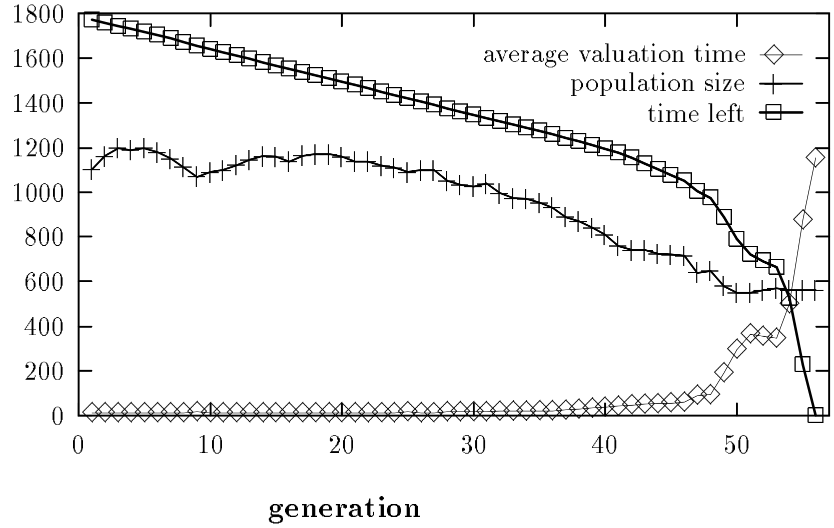

Figure 5.2: Plot of the average evaluation time (milli-seconds), the population size, the CPU time still available (seconds) versus the generations.

was set to zero. Over 20 experiments of 1800 seconds on a sun4 workstation, the GA always finds a solution for the parity function of size 50 (51 inputs). On fig. 5.3, the vertical axis that corresponds to the fitness space can be interpreted as a graduation of the information learned about the parity. At generation 15 the GA solves the parity of size 1 (XOR), at generation 37, it reaches size 5. An analysis of the solution found at this stage shows that the best chromosome generates a family of neural nets N_L that solves the parity problem i.e, such that N_L computes the parity of $L+1$ inputs. But the number of units of the neural nets N_L is proportional to 2^L, whereas the number of visible units is proportional to L. The ratio of these two number goes to infinite, it overcomes the threshold for $L = 6$, so that the neural net N_L, $L > 5$ have a null fitness. This is why the fitness does not go to 50. At generation 47, the GA produces a family (N_L) of neural nets having only $O(L)$ units, that solves the parity problem. The GA has learned the recurrence. The fitness is in fact infinite, but remains 50 only because the computer has finite memory.

Fig. 5.4 shows the chromosome found by the GA. The corresponding L^{th} neural network solves the parity of $L+1$ inputs with $5L-1$ hidden units and $10L$ connections. It is clearly a combination of L neuronal groups each performing a XOR. Despite the low weight complexity –usually it is quadratic in the number of input units– L hidden units can be removed without any damages. So the solution can still be fault tolerant. We designed a software that allows to see the development of the network graph as well as the final encoded neural network. Figure 5.9 shows a picture generated by this software. It represents a neural net for $L = 20$ that solves the parity with 21 inputs. Dashed lines denote a weight -1, straight lines, a weight 1, a black disk represents a threshold 1, a circle represents a threshold 0. The raw solution has 41 input units, it produces the right neural network modulo the elimination of the 20 last input units with the process described in section 5.2.

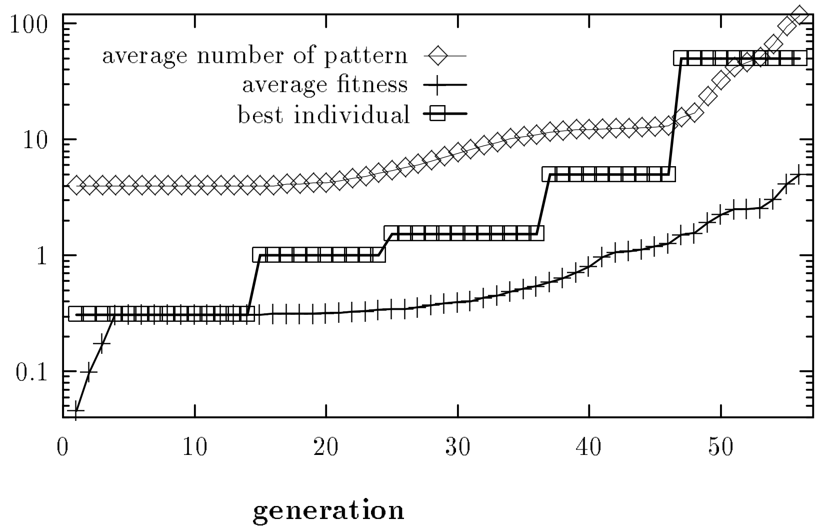

Figure 5.3: Plot of the average number of patterns used to evaluate the fitnesses, the average and best fitness in the current population.

5.4 Simulation with the symmetry

A close analysis of the solutions found by the GA for the parity problem shows that this problem is particularly easy to solve compared to other boolean problems. There are two reasons for this. First, it is possible to encode a family of neural networks that solves the parity with only 11 nodes. The corresponding code is shown in fig. 5.5 (c). The second reason is not obvious. In fig. 5.5 (a) we show a chromosome that develops the network (b). This network computes the XOR. The XOR is also the 2-inputs parity which is the problem of size one that the GA tries to solve in the preceding set of experiments. So the chromosome (a) develops the right neural network N_1 of all the family (N_L). It has therefore a fitness greater than 1. In fig. 5.5 (d) we develop the network-graph encoded by the code (c), with the following convention: when a reading cell reads the recurrent operator `REC`, it "freezes" its development. It means that it keeps its reading head on the node labeled by `REC` and does not rewrite itself any more. The network-graph (d) is almost the same as (b) except that one of its input unit is a reading cell that will give birth to a subnetwork computing the parity of lower order. The network-graph (d) makes clear the recursive process. It expresses the following recursive rule: in order to compute the parity of $L+1$ bits, compute the XOR of the first bit with the result of the parity of the last L bits. Chromosomes (a) and (c) differ only at one position which is `WAIT` in (a) and `REC` in (c). Since the chromosome for the problem of size one and the chromosome for the problem of arbitrary size are almost the same, this makes easy the GA search. The GA first found a neural net that look like (b) in order to solve the problem of size 1, and then, with a well placed mutation, the GA introduces a single recurrent operator `REC` that allows to go from (a) to (c) and to achieve the recurrence. We can express this in another way: Clearly, a CE develop a neural network by using a recurrence. It involves solving the problem of size one and the problem of the recurrence which is to find a way of building a network for the problem of size $n+1$ by combining network for the problem of size n with some supplementary neurons. In the case of

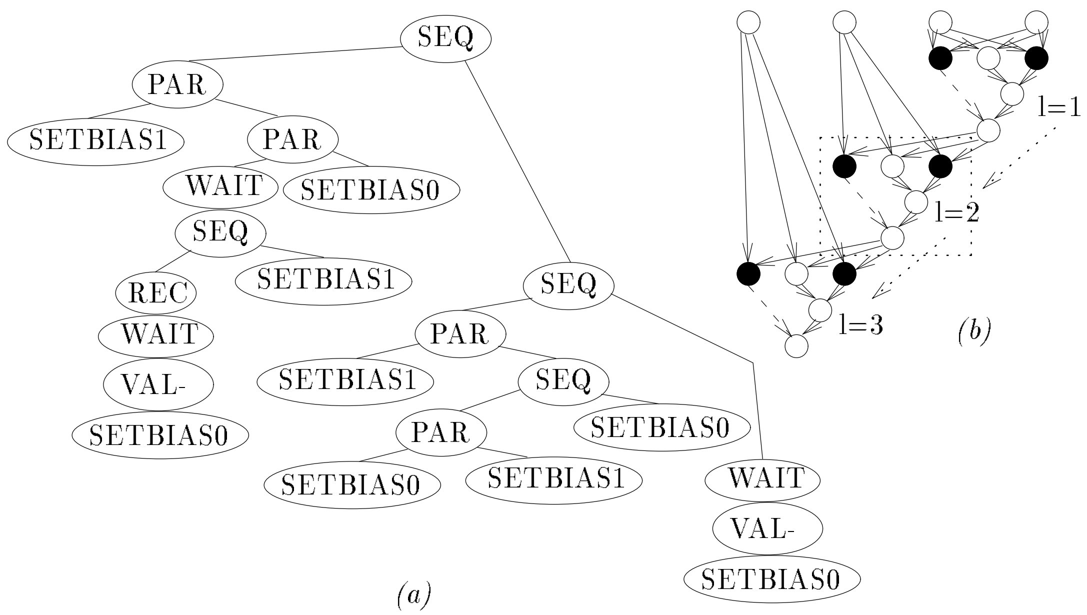

Figure 5.4: Solving the parity: (a) The chromosome found by the GA (b) the neural net for $L = 3$. The box in dotted line indicates a repeated neuronal group. — weight +1 • threshold +1 - - weight -1 ∘ threshold 0

the parity, the problem of size 1 and the problem of recurrence are almost the same, because both of them use a XOR neural network.

In order to tackle other boolean problems that do not have this particular feature, we had to extend the basic CE. We introduce A 2-ary branching operator denoted `BLIFE`. A cell that reads `BLIFE` executes the following algorithm:

```
1- If (life > 1) reading-head := left subtree of the current node
2- Else  reading-head := right subtree  of the current node
```

This operator gives account for the fact that the computation of the problem of size one can have nothing to do with the computation of the recurrence.

Whe now describe an experiment where the problem of size 1 and the problem of recurrence are not the same, it is the symmetry problem. Table 5.2 summarizes the key features of the problem for evolving a CE that generates a family of Neural network for computing the symmetry function of arbitrary large size. The GA tries to find the code for a family of neural network (N_L) where N_L computes the symmetry of $2 * L$ inputs. if the input is symmetric, the neural net must output 0, and 1 otherwise. The solutions are tested for the symmetry problem with up to 40 inputs. The program-symbol { `END PAR SEQ WAIT VAL+ VAL- INCLR DECLR INCBIAS DECBIAS CUT` } have been described in chapter 2. After many unsuccessful trials, we noticed that most of the time, the raw neural nets produced by the GA had not the right number of input or output units. They were evaluated modulo the convention described in section 5.2. We added a term in the fitness that rewards neural nets having the correct number of visible units. Doing so, we hoped that two kinds of individuals would coexist: Individuals developing neural nets with good mapping but not the right number of visible units, individuals developing architectures having the right number

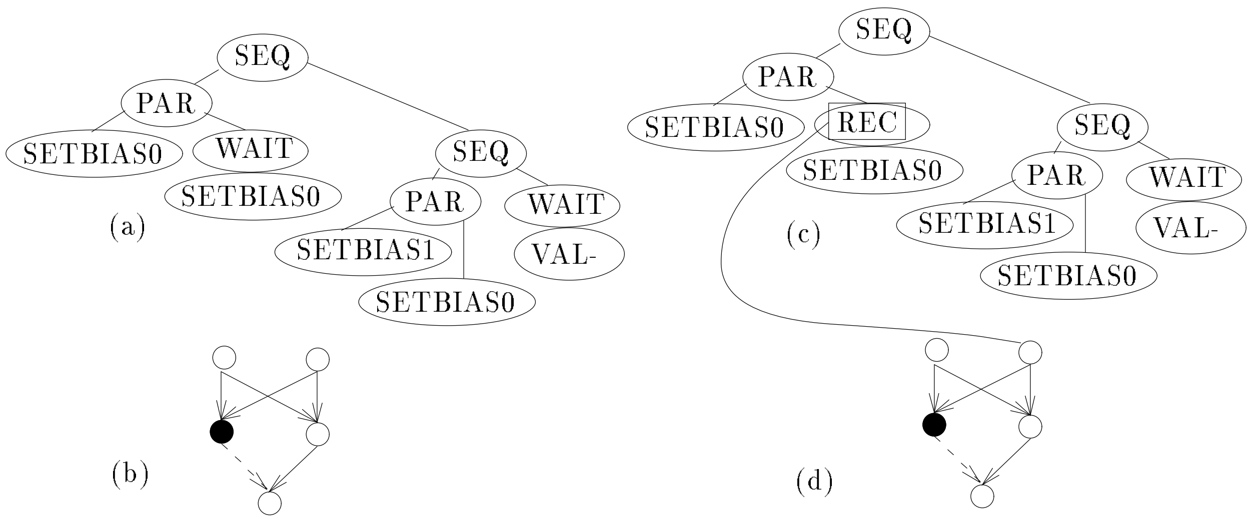

Figure 5.5: (a) The chromosome for the xor (b) the corresponding neural net (c) a chromosome for the parity function (d) The corresponding neural net, with the developement of the recursion frozen. — weight +1 • threshold +1 - - weight -1 ∘ threshold 0

Objective:	Find the CE of a neural network family that computes the symmetry of an arbitrary large size
Terminal set:	`END`
Function set:	`PAR SEQ WAIT VAL+ VAL- INCLR DECLR INCBIAS DECBIAS CUT REC BLIFE`
Fitness cases:	the number of fitness cases is determined by the GA For each developed neural net: If the neural net have less than 6 inputs the whole input space is tested otherwise an equal number of symmetric and non symmetric random patterns of bits are tested.
Raw fitness: of one neural net	The mutual information between the function computed by the neural net and the target function
Standardized fitness of one neural net	raw fitness divided by the information of the target function It ranges between 0 and 1
Standardized fitness	The sum of standardized fitness of the developed neural net. It ranges between 0 and 20
Parameters:	$T = 10800s, t_m = 0.005, l_0 = 30, l_{max} = 50$
Success Predicate:	The standardized fitness equals 20. That is the 20 first networks are developed and their fitness is maximum

Table 5.2: Tableau for genetic search of a neural network family for the symmetry problem up to 21 inputs, using CE

of visible units, but badly weighted, with bad mapping. The recombination between both kind of individuals should produce interesting offsprings. What happens is that this apparently minor modification in the fitness actually brought the GA to success, where it was previously constantly failing. More precisely, the new fitness is a weighted sum of the old one plus a term t. If the number of input (resp. output) unit is right, one adds 0.5 to t. In the simulation we used: new fitness = old fitness$*0.85 + t*0.15$. The new fitness still lies between 0 and 1.

The symmetry problem is much more difficult than the parity, because the recurrence requires a different computation than the problem of size one. The problem of size one is a XOR with two inputs. To make a recurrence, as an example, one can use the network shown in fig. 5.6 (b), where the development of cell reading a `REC` operator is frozen. The network in fig. 5.6 (b) expresses the

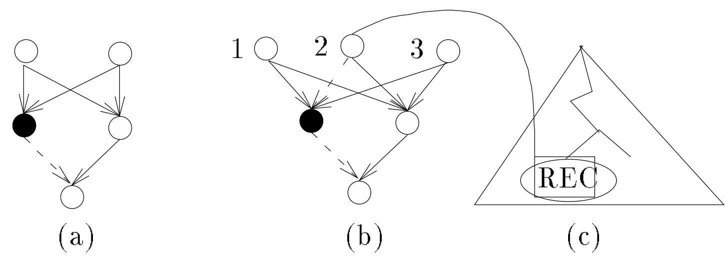

Figure 5.6: The symmetry boolean function, (a) a neural network for the problem of size 1, (b) A network-graph for the recurrence problem, (c) Sketch of the grammar tree,

following recurrence: A vector of n bits is not symmetric if either the middle $n-2$ neurons are not symmetric, or the first and last neuron are in a XOR configuration. In fig. 5.6, the network-graph (b) is clearly different from the network-graph (a), because the problem of size 1 is different from the recurrence problem. Therefore the GA must learn two distinct tasks instead of one. Recall that during the first generation, only the neural net N_1 is developed. Thus, the GA only faces the problem of size one. A few generations later, when the chromosomes are good enough, the networks N_L for $L > 1$ are developed, and GA faces the recurrence problem. Hence, the GA faces a changing environment. It is a difficult problem, since the population can converge prematurely toward a solution fitting the first environment. For this reason, we left the sun4 workstation for Four nodes of an IPSC860. Each node of this machine has approximately the power of one and a half sun4. The same previous GA run on each node. For one hundred births, the GA exchanges one individual with the neighbors. This individual is the best over a random subset of all the population. Since asynchronous communication was simulated, the communication time is negligible. The parallel GA is studied in chapter 8.

Recall that the GA deduces the population size p and and the number of generations g out of a function that yields the expected number of generations given a population size. We take $g = p/50$ in this simulation. Over 9 experiments of 3 hours each the GA finds a solution in 6 of the trials for the symmetry function of size 20 (40 inputs). We now report a successful trial. The population size decreases from 6000 to 1400. The average number of patterns allocated increases from 4 to 1800. On fig. 5.7, the vertical axis that corresponds to the fitness space is a graduation of the information learned about the symmetry. At generation 5 the GA solves the symmetry of size 1 (XOR), at generation 17, it solves the symmetry of size 2, with an approximation of the recurrence. At generation 20, the GA produces a family (N_L) of neural nets having only $O(L)$ units, that solves the symmetry problem. The GA has learned the recurrence.

The chromosome found by the GA is: `VAL−(SEQ (SEQ (PAR(END )(WAIT(BLIFE(PAR(REC(END ))( VAL+(VAL+(WAIT(VAL+(END)))))) (DECLR(BLIFE(INCLR (VAL+( WAIT(VAL−(END )))))(DECLR(INCLR(END )))))))) (PAR(INCLR(DECLR(END )))(INCBIAS(INCLR(BLIFE(VAL−(END )) (REC(BLIFE(END )(VAL+(WAIT(BLIFE(END )(END ))))))))))) (DECLR(VAL+(INCLR(−(END ))))))` In this chromosome a lot of information is not useful. Keeping only the genes that modify the network−graph, it is possible to simplify it. Fig. 5.8 shows the simplified chromosome, along with the resulting neural network for $L = 3$, we use the same convention to draw the network, as in fig. 5.4. The simplified chromosome has 24 genes, it is shorter than the solution we found by hand. To design a solution by hand, the natural way is to put an operator `BLIFE` at the root of the tree, to encode a XOR subnetwork in

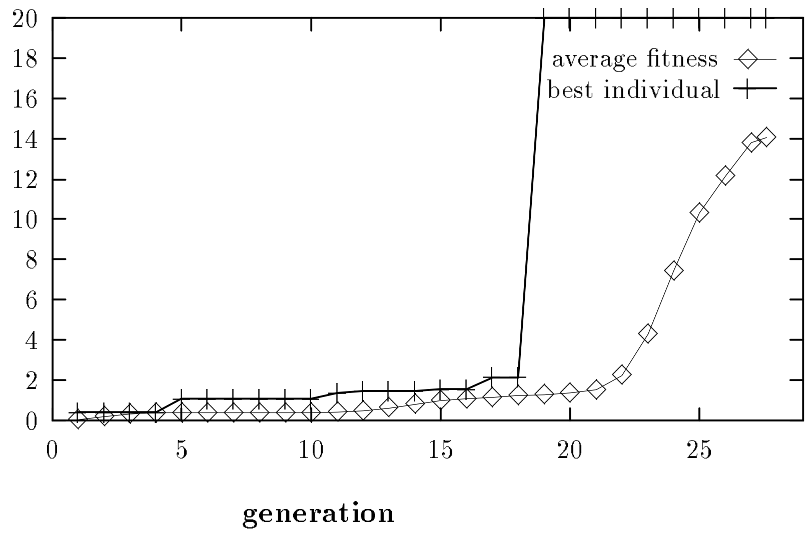

Figure 5.7: Plot of the average and best fitness in the current population versus the generations

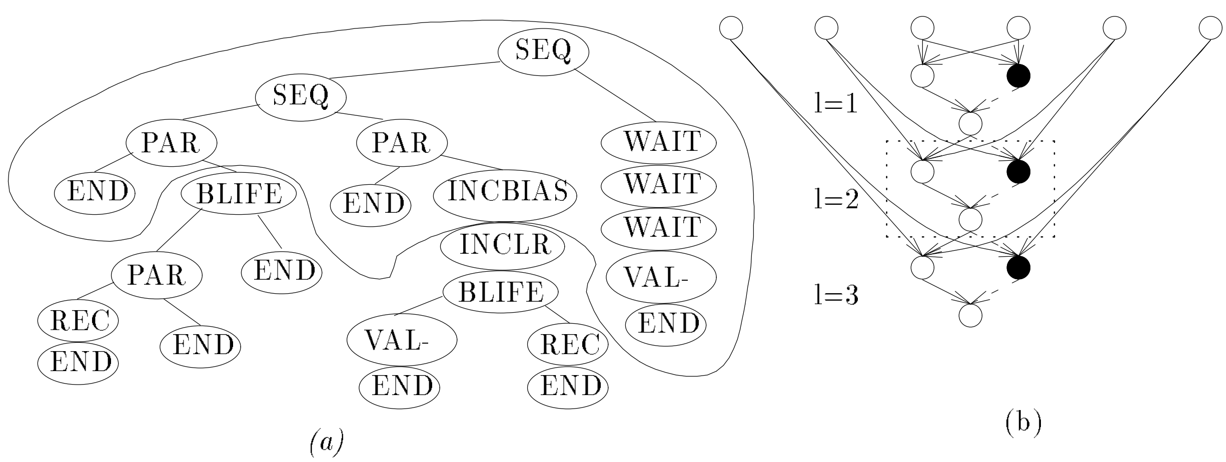

Figure 5.8: Solving the symmetry: (a) The chromosome found by the GA (b) the neural net for $L = 3$. The box in doted lines indicates the repeated neuronal group

the right branch, and a subnetwork doing the mapping in fig. 5.6 (b) (for the recurrence) in the left branch. This gives the 29 genes chromosome: `BLIFE(SEQ (PAR(PAR(END )(REC(END )))(END ))(SEQ (PAR(INCLR(INCBIAS(VAL−(END )))) (END ))(WAIT(VAL−(END )))))(SEQ (PAR(END )(END ))(SEQ (PAR(INCBIAS)(END ))(WAIT(VAL−(END ))))` The GA was able to concentrate information using the following trick: It produced a 13 genes piece of code (encircled in fig. 5.8 (a)) common to both the XOR subnetwork and the subnetwork for recurrence and uses the operator `BLIFE` twice at positions where the XOR subnetwork and the subnetwork for recurrence differ. The corresponding L^{th} neural network solves the symmetry of $2L$ inputs with $3L-1$ hidden units and $8L-2$ connections. We believe it is an optimal solution. It is clearly a combination of L subnetworks. It is similar to the network-graph found by hand, depicted in fig. 5.6. Figure 5.10 represents the neural net for $L = 20$ that solves the symmetry with 40 inputs.

5.5 Conclusion

This chapter shows that the Genetic Algorithm using Cellular Encoding is able to automatically and dynamically decompose a problem into sub-problems, using a kind of divide and conquer strategy, generate a sub-neural network that solves the sub-problem, and rebuild a neural network for the whole problem, using copies of that sub-network. Clearly, it is not possible to solve a problem like the 50 input parity with a gradient learning such as backpropagation, because this kind of learning does not scale well with the size of the problem, as Mulhlenbein points out in [23]. So this chapter clearly show the superiority of the GA+CE technology in the particular case of boolean functions that exhibits a certain amount of regularity.

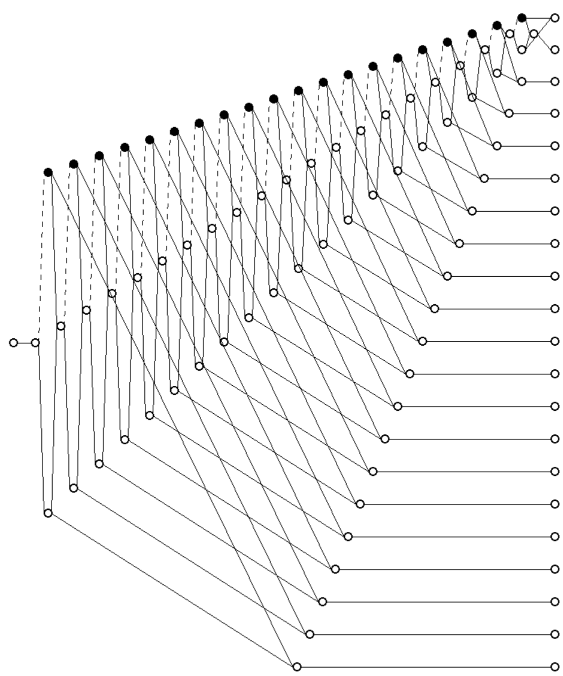

Figure 5.9: A neural network for the parity of 21 input units.

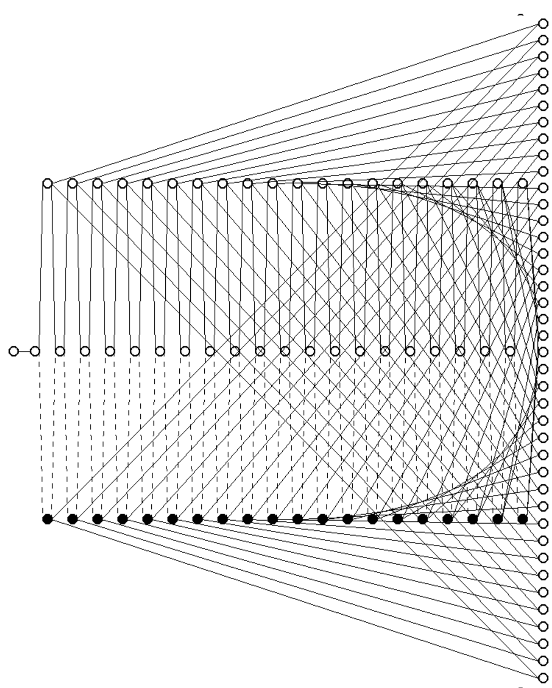

Figure 5.10: A neural network for the symmetry of 40 input units.

Chapter 6

Interaction between learning and evolution

6.1 Introduction

The difficulty with combining the search of neural network lead by the GA with a gradient learning is the high cost of each evaluation. If we must run a back-propagation algorithm (or some faster, improved form of gradient descent) for the evaluation of each neural network, the number of evaluations needed to find improved networks architectures quickly become computationally prohibitive. The computation cost is typically so high as to make genetic algorithms impractical except for optimizing small topologies. For example, if an evaluation function for a modest-sized neural network architecture on a complex problem involves one hour of computation time (which maybe unrealistically low for a hard problem), then it requires *one year* to do only 9,000 architecture evaluations in a genetic search. If the architecture is indeed complex (e.g. 500 connections or more) 9,000 evaluations is likely to be inadequate for genetic search.

The research presented in this chapter focuses on the addition of learning to the development process and the evolution of grammar trees. We explore alternative directions to the approach described before. We propose directions which need lighter computation cost. The genetic algorithm is not used to find architectures adapted to a particular learning algorithm. Instead, learning is used to speed up the genetic algorithm. We compare the speed up obtained with different possible ways of combining learning and the genetic algorithm. In particular, the following four modes of learning are explored: 1) using a basic genetic algorithm without learning, 2) using simple learning that merely changes the fitness function by making weight changes on simple networks, where the fitness function measures the raw network performance (called Baldwin Learning). 3) using a more complex form of learning that changes weights on simple networks, then "reuses" these weight changes in the development of larger nets (called Developmental Learning), and 4) using a form of Lamarkian evolution where learned behavior is coded back onto the chromosome of an individual so as to be passed along to its offspring during genetic recombination (called Lamarkian Learning).

We can make a feasible test of Lamarkian Learning, Baldwin Learning and Developmental Learning only because we use Cellular Encoding to encode our neural nets. Three features of cellular encoding are necessary: CE encodes both the architecture and the weights, With CE it is possible to code back a weight change on the grammar tree. CE is recursive. The first point allows to study Baldwin Learning, the second point: Lamarkian Learning and the third point makes it possible to study Developmental Learning.

Section 2 discusses the role of learning in evolution and the various forms of learning and combined learning-evolutionary search strategies to be used in these experiments.

6.2 Adding learning to cellular development

One way to speed up the search for functional networks is to add some form of learning to the developmental process. In general, to be effective the learning process must be less costly than reproducing and testing a new individual or, if the cost of learning approaches or exceeds that of reproducing and testing a new individual, learning must provide a greater chance of resulting in improved behavior.

There are numerous ways in which learning could be added to the developmental process and several problems that must be considered. First, we are working with Boolean networks. There are several reasons why the learning algorithm would ideally result in Boolean weights. One reason to learn only Boolean weights is that the learned information could be coded back into the grammar. Coding the learned weights back into the grammar is useful if some form of Lamarkian evolution is employed. By Lamarkian evolution, we mean that chromosomes (grammar trees) produced by recombination are improved via learning and these improvements are encoded back onto the chromosome. The improved chromosomes are then placed in the genetic population and allowed to compete for reproductive opportunities.

An obvious criticism of Lamarkian evolution is that it is not Darwinian evolution: it is not the way that most scientists believe evolution actually works. Another criticism is that *if* we wish to preserve the schema processing capabilities of the genetic algorithm, then Lamarkian learning should not be used. Consider the simple case in which chromosomes are bit strings instead of grammar trees. Changing the coding of offspring's bit string alters the statistical information about hyperplane subpartitions that is implicitly contained in the population. Theoretically, applying local optimization to improve each offspring undermines the genetic algorithm's ability to search via hyperplane sampling. The objection to local optimization is that changing information in the offspring inherited from the parents results in a loss of inherited schemata, and a loss of hyperplane information. Selection changes the sampling rate of schemata representing hyperplanes.

Thus, hybrid algorithms that incorporate local optimizations typically result in greater reliance on hill-climbing and less emphasis on hyperplane sampling. This should result in a less global exploration of the search space, since it is hyperplane sampling that gives the genetic algorithm its global search ability. Despite this theoretical objection, hybrid genetic algorithms typically do well at optimization tasks for several reasons. First, the hybrid genetic algorithm is hill-climbing from multiple points in the search space. Unless the objective function is severely multimodal it may be likely that some strings (offsprings) will be in the basin of attraction of the global solution. Second, a hybrid strategy impairs hyperplane sampling, but does not disrupt it entirely. For example, using learning or local optimization to improve the initial population of strings only biases the initial hyperplane samples, but does not interfere with subsequent hyperplane sampling. Third, in general hill-climbing may find a small number of significant improvements, but may not dramatically change the offspring. In this case, the effects on schemata and hyperplane sampling may be minimal.

6.2.1 The Baldwin Effect

There are other ways that evolution and learning can interact that is not Lamarkian in nature. One method which has recently received new attention is the *Baldwin effect* [10] [1]. From an implementational point of view, exploitation of the Baldwin effect requires that individuals employ some form of learning, but the acquired behavior is not coded back to the genetic encoding as in Lamarkian learning. Instead, the learned behavior affects the objective function; in other words, fitness is a function of both inherited behavior plus learned behavior. The inherited behavior refers to the features directly encoded in the genes. From this point of view, learning changes the fitness landscape. In particular, if the genetic encoding (or, more precisely, the phenotypic behavior associated with a particular genetic encoding) could be significantly improved by a few minor changes, then any form of learning that "corrects" the inherited behavior also changes the fitness landscape associated with the objective function so as to reward solutions that are close to a more highly fit *behavioral* solution that can be reached via learning. The idea that learned behavior could influence evolution by creating selective pressure toward genetically encoding or genetic predisposing learned behavior was first proposed by J.M. Baldwin [13] almost a hundred years ago.

Hinton and Nolan [10] offer the following simple example to illustrate the Baldwin effect. Assume that the fitness landscape is flat, with a single spike representing a target solution. Also assume that individuals can recognize when they are close to a solution and can alter their behavior to exploit the target solution. As pointed out by Hinton and Nowlan, it is important that individuals be able to recognize "improved" behavior. The closer an individual is to the target solution, the more likely that individual is to learn the target solution. Learning, in this situation builds a basin of attraction around the spike, so the fitness landscape is transformed into a less difficult optimization problem.

Hinton and Nolan acknowledge that this example is extreme and unrealistic, but it does serve to illustrate the principles behind the *Baldwin effect.* They also illustrate the Baldwin effect using a genetic algorithm and a simple random learning process that develops a simple neural network. The genetic encoding specifies a neural network topology by indicating which of 20 potential connections should be used to connect a given set of artificial neurons. The objective function (without learning) is such that the "correct" network results in increased fitness, and all other networks have the same, inferior fitness. This creates a spike at the "correct" solution in an otherwise flat fitness landscape. The genetic encoding uses a 3 character alphabet (0, 1, ?) which specifies the existence of a connection (i.e., 1), the absence of a connection (i.e., 0) or the connection can be unspecified (i.e., ?); if the connection is unspecified it can be set randomly or set via learning. Learning in this case is merely a random search of possible assignments to unspecified connections.

Two interesting results are reported by Hinton and Nolan. First, genetic search without learning fails to find the solution; this is not surprising, since the "solution" is a needle in a haystack. Second, the final solutions that are found by genetic search with learning are left somewhat underspecified. In other words, a small number of unspecified (?) characters will remain in the genetic encoding after the solution is close enough to the target solution so that it is reliably found via learning. (In theory, this second phenomenon should not happen; selective pressure coupled with recombination should continue to drive out underspecified alleles. The underspecification found by Hinton and Nolan might have occurred because the search was prematurely terminated.)

Hinton and Nolan's work illustrates the Baldwin effect, but provides little insight about whether the Baldwin effect can be intentionally exploited to accelerate evolutionary learning. Belew [3] also offers a good critique of some of the assumptions underlying the Hinton and Nolan model. In particular, the number of learning trials allocated to the individual must be sufficient to occasionally locate the spike; otherwise, it remains a needle in a haystack.

In the current context we would like to exploit some form of learning that can correct the specification of a network made by the grammar tree corresponding to a particular cellular encoding. We then wish to test the hypothesis that learning (which is used to change the value returned by the fitness function, but which does not alter the genetic code) will result in a genetic search that more quickly finds networks to solve the parity and symmetry problems.

6.2.2 Developmental Learning

Finally, there is still yet another way learning can be used in the current context that is intermediate between Lamarkian learning and directly learning weights. We will refer to this intermediate form of learning as *developmental learning.* This form of learning exploits a unique characteristic of the recursive encoding used by the cellular encoding method. Since the encoding allows for recursive moves in the grammar tree, learning on Network N_1 which occurs for $L = 1$ can be exploited for the development of networks N_L for values of L greater than 1 ($L = 1$ results in one iteration of development with no recursion). In other words, learning can actively take place for iteration $L = 1$. This can be accomplished with relatively minimal cost, because network N_1 is very small due to a short development. Learning should only modifies a few weights, one or two weight in our experiments. The learned information can be coded back into the grammar, so that the modified grammar generates the modified weights. The modified grammar can then be used to develop larger nets N_L where $L > 1$. This may help under an "hypothesis of regularity" that states:

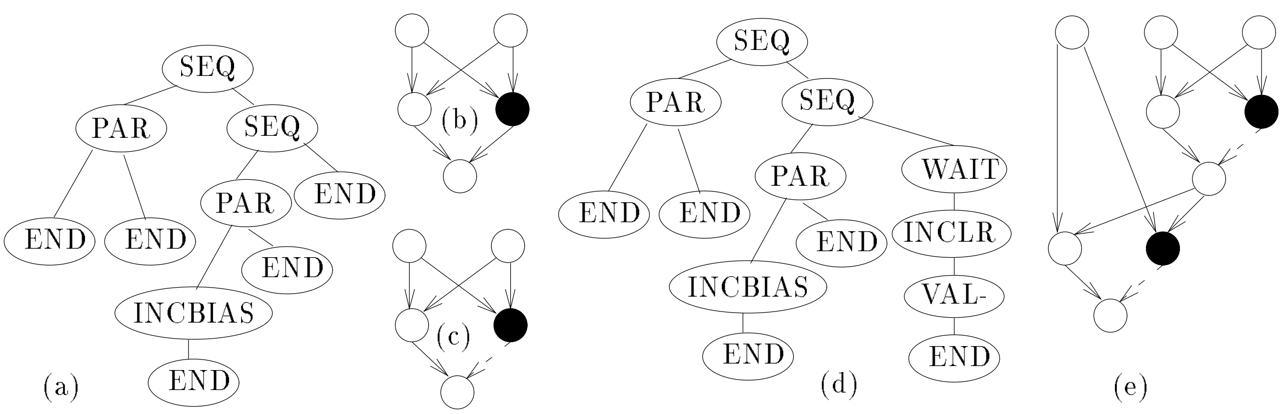

Figure 6.1: Algorithm for the developmental learning: (a) Chromosome produced by the GA, (b) Development of neural net N_1, for a life $L = 1$, (c) After the training of N_1, a weight feeeding the output unit has been modified to -1 (dashed line). (d) Back coding on the chromosome, (e) Development of neural network N_2 using the modified chromosome.

If a grammar develops N_1 right, it is more likely to develop N_L right, for $L > 1$.

This hypothesis is valid only is the function that N_L must compute is itself regular, like parity or symmetry. Figure 6.1 illustrates how the grammar can successfully register a weight modification for $L = 1$, and replicate it 2 times for $L = 2$. The network N_2 represented in figure 6.1(e) has two weights set to -1. In this example, the network N_L would have L weights set to -1. The method used for back-coding is explain in section 3.5.

The way in which less-developed nets require less computational effort to evaluate may be important from a biologist point of vue. We did not collect statistics on this, the reason of this lower cost is clear: the size of N_L grows with L. If there is k recurrent program symbol `REC` the size will grow as k^L, and in the case $k = 1$, the size will grow linearly. And the evaluation time grows with the neural network size.

Developmental learning as defined above is not Lamarkian in the sense that the learning is coded back onto the chromosome for developmental purposes, but any subsequent offsprings that are produced by that individual do not carry the acquired information in their encodings. What is actually happening is that the genetic code is used to store learned information during the "lifetime of the individual" as development of different networks occurs, but this learned information is not directly passed on during reproduction. We refer to this as *developmental learning*.

We are not claiming biological plausibility for this method, since we assume there is no biological evidence for using recursive cellular encodings in the exact way they are used here. However, there are species that do code learned information onto their DNA as a way of storing learned information; this storage is local and does not impact reproduction.

It should also be pointed out that *developmental learning* works, if it works, because it is also exploiting the *Baldwin effect*. In other words, developmental learning is just a more effective way of learning, a more effective way of exploiting behaviors acquired during the lifetime *and development* of the individual. The resulting behavior is not passed on to offspring.

6.2.3 Comparative Goals

Our goals, then, are to examine Lamarkian learning, the impact of the Baldwin effect on learning, and the use of the developmental learning method. For Lamarkian learning and *developmental learning* it is necessary to recoding the learned behavior back onto the chromosome. Since cellular encoding specifies a Boolean network, the learning algorithm we use should also result in a Boolean network. In addition, we would like the learning process to be very inexpensive. In the case of exploiting the Baldwin effect, the analogy suggested by Hinton and Nolan also suggests a fast, relatively minimal form of learning: learning should provide information that tells you that you are close to a solution. More specifically, an ideal learning algorithm for the set of experiments we have outlined would be a one-pass learning algorithm that returns one or two minor changes in the network that could potentially lead to improve performance. These changes can then be tested.

Given the requirements we have outlined, we chose to use Hebbian learning. This also means that we will use learning in a somewhat restricted way.

6.2.4 Implementation of the Hebbian Learning

Assume that we have a Boolean neural network with one hidden layer where it is known that the first layer has a correct set of weights and hidden units have correct thresholds. Clearly, this implies that the problem has been reduced to learning a linearly separable function with respect to the hidden units. Perceptron learning might be used at this point, except this algorithm does not result in a Boolean net and it is not a one-pass learning algorithm. If we limit learning to a single pass, however, we can look at the activations of both hidden nodes and output nodes and collect information about whether the connection should be inhibitory or excitatory between any particular hidden unit and output unit. That is, for each training pattern and each pair of hidden-output units, indicate the weight should be positive (i.e., $+1$) if both the hidden unit and output unit is "on" and indicate the weight should be negative (i.e., -1) if the hidden unit is on and the output unit is off. (We do not consider cases where the hidden unit is off, since these cases do not effect the activation of the output unit in a feed-forward network.)

Recall that the weights of our neural nets are Boolean (± 1). For the parity and symmetry applications the neural nets have a single output unit. We used the following algorithm to learn new weights for the links that feed into the output units of the neural net. Each training pattern in presented to the input units. The activation level of each hidden neuron is computed, until all hidden units are processed. The desired output value is then clamped to the output unit. For each link l feeding the output unit, the following Hebbian information is computed. Each link l has a variable d_l. If the activities of the units linked through l are the same, d_l is incremented; otherwise d_l is decremented. The variable d_l is a correlational measure of the activities of the neurons linked through l.

Before the learning, the variables for each d_l are initialized to zero. Note that this is somewhat different from perceptron learning in as much as we collect statistical information over all of the training patterns, not just those that result in errors. This learning also is similar to the local learning used in Boltzman machines [5], however, in these experiments we are not concerned with escaping local minima and have no need for the simulated annealing component that is used in the Boltzman machine to achieve a global search of weight space.

After processing all the patterns, the variable d_l is used to determine whether to flip the weight w_l or not. We consider the subset of links l for which d_l and w_l have an opposite sign (Recall that $w_l = \pm 1$). We will modify only the weights of the links in this subset. These are the links for which the weight is different from the correlation, and the aim of hebbian learning is precisely to set the weights equal to the correlation. It is the reason why we claim to do an Hebbian-like learning. For these links, $|d_l|$ can be considered as a measure of the badness of the weight, because it is the amount of change to make in order to bring back d_l to the right sign. Hence, we sort the links of the subset according to the quantity $|d_l|$. The first link, that is, the one with the biggest $|d_l|$ is flipped with a probability 1. The second link is flipped with a probability 0.05. In general, the i^{th} link is flipped with a probability $(0.05)^(i-1)$. This learning algorithm ensures that at least one link will be flipped most of the time. The only requirement is that at least one link l has d_l and w_l of opposite sign. Under this condition, the average number of flipped link lies between 1 and $1/(1-0.05) \approx 1.05$. After the weight changes are made we process all the training data again. We accept a set of weight changes only if the fitness has improved. Only one epoch of training is used.

Less than two links are flipped on average. This should be sufficient to create a Baldwin effect in as much as learning should "build shoulders" around minima in the fitness landscape associated with the raw evaluation function (i.e., without learning). In our experiments, bias terms (i.e., threshold values) are not learned and must be found by genetic search.

Experiments show that this learning, although very simple, is sufficient to flip the correct weight in a exclusive-or (XOR) network where all the weights are 1, with two hidden units, where one hidden unit computes the logical AND and the other one computes the OR. In order to get a XOR one must set the weight between the AND unit and the output unit to -1, so that the output units outputs the results of the OR unit, unless the AND unit is activated in which case it outputs 0. Thus in the case of XOR, and in fact all parity problems (of which the XOR function is an example), the limited form of Hebbian learning which we use can exploit a fortuitous feature of the problem. We will show, however, that such fortuitous features need not be present for this learning strategy to be effective.

A famous work done by Hinton Ackley and Sejnowsky, about the Boltzmann machine has shown that hebbian learning tries to optimize the mutual information between the function computed by the neural net and the target function. Since the fitness we used is also based on mutual information, the GA and the learning tries to optimize the same thing.

6.2.5 Operationalizing Developmental learning

One of the advantages of the recursive encoding scheme is that evaluation can be done in an incremental fashion. For parity or symmetry, a network can be evaluated with respect to its ability to solve a 2 input case. If a solution to the 2 input case is found, the next recursive iteration of development can be carried out and the 3 input case can be evaluated. In this way, evaluation is fast early in the evolutionary process and becomes more expensive as networks become more proficient at handling the lower order input cases.

Analogously, we use learning only on the neural network during the first iteration of the recursive development process. This neural net is the smallest of a whole family of networks that can be generated by a chromosome; it is smallest both in term of the number of units and the number of weights, so little time is expended on learning. But we would also like the other neural nets to benefit from the learning process. To do this, we must encode the weight changes that have occurred during the learning of the first neural net back to the chromosome using a combination of appropriate program-symbols; we can then develop the entire family of associated networks so as to exploit learning via the modified chromosome. During the development of another network, when a cell of the intermediate network graph executes the new program-symbols added by learning it will set its weights accordingly, thereby reproducing the same subnetwork as the one contained in the first neural network after learning.

One requirement of back-coding learning onto the chromosome is that the new code should allow us to correctly develop the first iteration neural network, including the modifications made by the learning. This is not always simple. Normally, when a cell becomes a neuron it executes the program-symbol `END`, and looses the reading head so that it can no longer rewrite itself. But in order to do the back-coding, we must keep the reading head. This reading head points to the precised node where the particular subtree of program-symbols should be inserted in order to accomplish a change of weight. A concrete example shown in figure 6.2 helps to illustrate this problem.

Consider the value of the link register, at the time the cell became a neuron. If this value was 1 and the learning changes the weight of the second input link into -1, then we generate the following subtree: `INCLR(VAL-(END))`. The program-symbol `INCLR` sets the link register to 2, the program-symbol `VAL-` sets the second input link to -1. Let N be the node pointed by the reading head of the neuron. This node has an arity of 0, and has previously transformed the cell into a neuron. In order to insert the new subtree, we must transform the leaf-node into an interface node of arity one. We replace the program-symbol `END` by the unary program-symbol `WAIT` for Wait, and then insert the subtree that adds in the learned weight. When a cell executes this program-symbol, it positions its head on the next subtree, and waits for the next rewriting step.

We cannot be sure that this type of simple change in the grammar-tree will exactly recode the learned behavior. The new encoding will produce the learned behavior if two conditions are satisfied:

- Learning is done only on the first neural net of the family.
- During the development with the modified code, for any cell C_r of the intermediate neural network that starts to read the inserted subtree, those cells that have fan-out connections that feed into cell C_r must have lost their reading heads. This ensures that the new changes affecting C_r do not interact with the development of those cells whose fan-out connections feed into C_r.

The first neural net of the family has the feature that no more than one neuron reads a given leaf of the grammar tree. This is not the case for higher order neural nets. If the first condition was

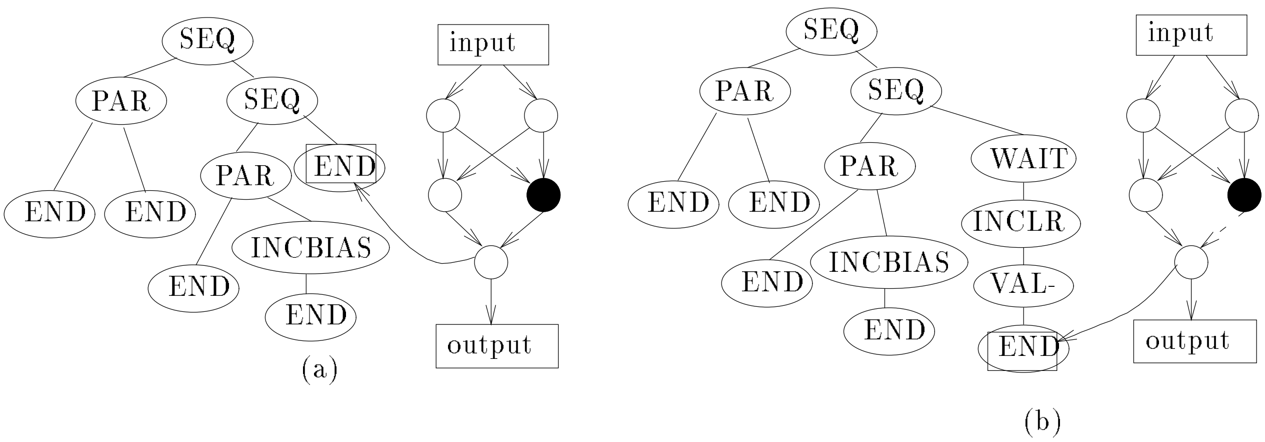

Figure 6.2: Back-coding on a grammar-tree, that builds the code of a XOR network. Illustration (a) is the net before back-coding. In illustration (b) the grammar tree has been modified and the corresponding weight change is shown. The modified network computes the XOR.

not met, two distinct neuron could read the same leaf, and both could try to concatenate a distinct subtree corresponding to their own weight modification at this leaf. It is not possible to insert two distinct subtrees at a single leaf. Moreover, these two subtrees could encode contradictory modifications of the weights.

If the second condition is satisfied then the change introduced by the added subtree will remain local. Otherwise it may produce unexpected side effects. For example, in Figure 6.2 assume the program symbol `INCBIAS` in the grammar tree were replaced by a `PAR` The corresponding node (which is black in Figure 6.2) will now split instead of changing its threshold and there will be three fan-in connections to the output node. Also assume we still wish to change the second fan-in connection to a negative weight, but not the third connection. If the inserted weight change executes before the parallel division, , then both the second and third fan-in connection will be negative. If the weight change executes after the parallel division, then only the second weight will be negative.

In order to ensure the proper sequence of development, we could add several waiting program-symbols `WAIT` to key positions so that the new inserted section of the tree executes in the proper sequence. However, this could result in an increase in the amount of time needed to develop the higher order neural nets; more critical, however, is the likelihood that it would make it more difficult to do Lamarkian learning because the chromosome could quickly become very big. This means that the encoding would no longer be quite as compact and recombination might have to process deeper and deeper grammar trees. Note that this strategy of adding long wait delays *does not* impact developmental learning, since the "expanded" chromosomes never enters the genetic population. Thus, there is no problem adding enough "wait" delays to ensure that all other development processes have terminated before effecting learned weight changes.

However, in the current experiments we have settled on the following compromise. By replacing the 0−ary program-symbol `END` by the unary program-symbol `WAIT` and then attaching the subtree which produces the weight change, we add at least one wait delay program-symbol, and hope that in this way, we sufficiently increase the probability that the second condition is verified. For reasons of consistency, we used this approach for both Lamarkian and developmental learning strategies. This does not guarantee that our encoding of learned behavior does not unexpectedly interact with the developmental process, but empirically we find that in most cases it should not.

6.3 Simulation of different possible combinations

We now describe the result of some experiments which compare the different ways of adding learning. The *basic GA* refers to the genetic algorithm without any learning. We have described the encoding in chapter 2. We remind the reader that this encoding encodes not only the architecture but also the weights. Thus, it is possible to run the GA without any learning. Once the neural net has been developed we perform Hebbian learning. If we use only the improved fitness of the trained network N_1 obtained with a life $L = 1$, (in this case $L = 1$, there is no recursion) we call it *Baldwin learning.* If we code the learned information back to the chromosome, and use it to develop higher order networks, that is *developmental learning.* If the improved chromosomes are placed back in the genetic population and allowed to compete for reproductive opportunities, the result is *Lamarkian learning.* The following experiments will compare these four learning modes on two different tasks of increasing complexity.

6.3.1 Strategic Mutation

Over the course of the experiments, we noticed that one of the most difficult problems faced by the genetic algorithm was to correctly place the recurrent program-symbol `REC`. In the case of the parity problem, a correct estimation of where to put a program-symbol `REC` is enough to solve the recurrence, as indicated by figure 5.5. On the other hand, an incorrect placement prevents the genetic algorithm from correctly developing any of the other networks in the family of networks except the very first one. This "all or nothing" feature causes a lack of smoothness in the fitness landscape which makes placing the `REC` program-symbol difficult. In order to overcome this problem we used a strategic form of mutation for the `REC` program-symbol. By mutating `REC` 10 times more often than other program-symbols, most `REC` program-symbols are quickly removed and, in general, allele `REC` is 10 times less represented than other program-symbol of arity 1. In order to offset this bias against `REC`, each program-symbol of arity 1 has a probability of $4 * t_{mut}$ to mutate into a `REC` program-symbol, where t_{mut} is the mutation rate of the other alleles. The combined effect of these mutations strategies is to explore a large number of different placements for `REC`

6.3.2 Parity

In this set of experiments, the GA was run for 900 seconds on one node of the parallel machine Ipsc860. To solve this problem, the genetic algorithm must find a grammar encoding that develops a family of neural networks (N_L), such that N_L computes the parity function of $L + 1$ inputs. The parity function returns the sum of all the inputs, modulo 2. We refer to table 5.1 for a description of the experiments parameters. The only parameter which changes is T which is now 900 seconds.

In order to produce a valid comparison, we ran 50 trials and computed the mean and standard deviation of two variables: the time required to find a solution, and the total number of neural nets evaluations. In order to obtain an estimation of the standard deviation of the result, divide by $\sqrt{50}$ the standard deviation of the variable. We report the standard deviation of the variable, to show that it is high, because we think that this information gives insight into the genetic search.

The parity problem is a little special with respect to developmental learning. The network N_1 developed with a life $L = 1$ implements the XOR. This XOR network has all weights equal to 1,

LEARNING MODE	TIME	time SD	EVALUATIONS	evaluation SD	SUCCESS
basic GA	240	197	23865	20317	36
Baldwin	234	205	14307	10699	28
Developmental	131	195	8404	8829	47
Lamarkian	66	85	5700	5220	49

Table 6.1: Initial Experiments for the parity problem which ran for 900 seconds on one node of the parallel machine Ipsc860 that is to say, with a single population. Success is a measure of how many times a solution was found for the parity problem. The number of trial is 50. The time is measured in seconds, throughout the chapter

except one weight that links the AND to the output unit, as shown in figure 5.5. Since we train only network N_1, and the weights are set to one by default, the learning need to change only weights that feed the output unit.

The first line of table 6.1 reports experiments with no learning, the next three use learning. The runs fall in two classes: some runs quickly find a solution, while others take a long time. This explains the high standard deviation observed in these experiments. The GA frequently get trapped at a suboptimal solution.

Although Baldwin learning does not improve the computation time, it needs 17000 neural network evaluations on average to reach success, whereas the basic GA needs 24000. This means that the average evaluation time is higher for the Baldwin learning. The average extra time needed to compute the Hebbian algorithm stays around 0.5 millisecond, whereas the average time required to develop and evaluate a neural network starts at 7 milliseconds and significantly increases through subsequent generations to finally reach 14 milliseconds. Therefore, the Hebbian learning algorithm takes less than 8 percent of the time taken to develop the neural net and evaluate it.

The average time required for the Baldwin learning is higher, because of the scheme used to evaluate the fitness of the code of a family (N_L) of networks. The basic GA does not typically develop the neural networks N_L for $L > 1$ in the early generations. For all of the experiments, the developing rule reported in the preceding chapter decides exactly when it is necessary to develop the neural network N_2, and then N_3 and so on, until $N_{l_{max}}$. When Hebbian learning is used, the neural net N_1 is quickly found. For example with the Baldwin learning on the output unit, the average time taken to find the first neural net is 19 seconds. The basic GA takes 176 seconds. Therefore with the Baldwin learning, the GA starts to develop network N_2 from the second generation; this has the side-effect of making each evaluation more expensive. Lamarkian and developmental learning do much better than both Baldwin and basic GA. Lamarkian is also better than developmental.

In order to solve the problem that some runs do not successfully find solutions, we used a bigger population and a parallel GA. The parallel GA that we used is described chapter 8. The number of experiments is still 50. For each experiment, the GA ran 3 hours on four nodes of the parallel machine Ipsc860. The population size ranges from 3500 to 2000 on each node. We therefore use a total of 12 hours of computation before deciding success or failure.

These results are reported in terms of computations per processor in table 6.2. Since 4 processors were used in these experiments, computation times and evaluations should be multiplied by a factor or 4. This GA is successful on all runs. The standard deviation is smaller, because the population is large enough to reliably sample critical subprograms of grammar trees. Baldwin learning now produces a speed up of 1.3 over the basic GA. Developmental and Lamarkian learning still produces

LEARNING MODE	TIME	time SD	EVALUATIONS	evaluation SD	SUCCESS
basic GA	173	64	17808	5818	50
Baldwin	135	86	12511	4799	50
Developmental	59	19	7160	1805	50
Lamarkian	55	18	6800	1763	50

Table 6.2: Second set of experiments for the parity problem which ran for 3 hours on four nodes of the parallel machine Ipsc860.

LEARNING MODE	TIME	time SD	EVALUATIONS	evaluation SD
basic GA	91	55	7332	3869
Baldwin	74	36	5811	2292
Developmental	48	20	4352	1840
Lamarkian	53	29	4381	2229

Table 6.3: Experiments for the symmetry problem which ran for 300 seconds on 32 nodes of the parallel machine Ipsc860.

better results. The effectiveness of developmental and Lamarkian learning are very similar.

6.3.3 Symmetry

The problem is to find an encoding that develops a family of neural networks (N_L), such that N_L computes the symmetry function of $2L+1$ inputs. The symmetry of $2L+1$ binary inputs returns 1 if and only if the input is symmetric and the middle bit is 1. This function has a fairly compact cellular encoding and is therefore not too difficult to learn with the genetic algorithm. The genetic algorithm runs 300 seconds on 32 nodes of an Ipsc860. The population size ranged from 300 to 200 individuals on each node, and between 9200 and 6400 on the combined set of processors. Table 6.3 reports an experiment with no learning, and the use of Hebbian learning on the output unit. The results are based on 30 experiments; every run finds a solution.

We indicate the computation time taken by each processor. Multiply by 32 to find the total computation effort. Developmental learning is slightly better than Lamarckian.

6.4 Conclusion

The addition of learning to the search process produced two interesting results. First, the notion of developmental learning was introduced. By allowing learning to affect subsequent development, early learning can be more effectively exploited and the high cost of learning on larger networks developed later in the "life" of the grammar tree can be avoided. In the current experiments, developmental learning also provides a fairer comparison between Lamarkian evolution and Darwinian evolution, that still exploits the Baldwin effect. Developmental learning allows the entire family of networks which can be developed from a cellular encoding to exploit learning. The same is true of the Lamarkian strategy, since offsprings that inherit learned behavior use it to develop all subsequent networks, not just those that correspond to networks developed by a single iteration through the grammar tree.

The second main finding of this chapter is that developmental learning appears to be competitive with Lamarkian learning, for the problem of finding a family of boolean neural networks. This is especially true on the symmetry problem. The results for the parity problem are more mixed, but the parity problem is unusual in as much as learning on only those connections that fan-in to the output unit is sufficient to find a correct set of weights. The use of learning (or local optimization) to change the fitness landscape without using Lamarkian learning strategies could be of general interest to that part of the genetic algorithm community concerned with general function optimization. Finally, the work reported here show that cellular encoding is a good and may be unique testbed for Lamarckian and developmental learning. The hebbian learning that we have used is very simple. Other forms of learning need to be considered. The next chapter will describe a learning that has been tailored for neural networks generated by the GA using cellular encoding.

Chapter 7

The Switch Learning

This chapter proposes a learning that especially fits networks generated by the genetic algorithm using cellular encoding. We first describe the features that such a learning should have. We then propose a learning algorithm that learns ± 1 weights. An extension of it learns weights in $\{0, -1, 1\}$, but is not very efficient. Finally, we report two kinds of experiments which demonstrate that the proposed learning fits the requirements. The first kind uses the learning on a fixed neural architecture, and study the convergence, and the speed of convergence, the second kind uses the learning in combination with the GA.

7.1 Specification of the learning algorithm

The proposed learning algorithm is designed to train networks generated by the GA using the Cellular Encoding. Therefore it has some specific requirements. The GA produces networks with integer biases and boolean weights. The learning should also find boolean weight, in order to be able to do backcoding of the learned information on the cellular encoding. Backcoding is required for Developmental learning and Lamarkian learning seen in chapter 6. We do not learn the bias because the cellular encoding of a bias is compact. Using a program-symbol like `INCBIAS`, there is no need to specify a link number, which is the case if we want to encode a weight-change. Hence the GA needs less effort to encode the right biases than the right weights. The neuron's sigmoïd is the step function. A neuron n computes its net input which is the weighted sum of the neighbor's activities. If the net input is lower or equal to zero, the neuron's activity is set to 0, else it is set to 1. The GA generates thousands of networks, so learning needs not be 100% successful on each network. On the other hand, learning must be fast, because the time of a GA run will be the learning time taken for one neural net multiplied by thousands. The purpose of the learning is to refine a neural net produced by the GA. It should change correctly a few weights in order to increase the performance of the network. It may find the global optimum, if the neural net before training is close to it. The networks generated by the GA using cellular encoding have some particularities. The neurons are sparsely connected, usually the fan-in is 2 or 3. The number of layers is high, so a standard back-propagation would fail. Most of the weights are 1, only a few weights are -1. This is because weights are set to 1 by default, and only negative weight values must be encoded, using program-symbol `VAL-`. The learning algorithm should take advantage of these particularities.

7.2 The Switch learning algorithm

```
Let c be the currently processed neuron
For each neuron n that fans into c through link l
  Try to modify n's activity; Compute the new net input;
  switch(Comp(new net input, old net input))
    case 0 : if(Small(old net input)) r_n := r_n + r_c;
             w_n := w_n + w_c;
    case + : r_n := r_n + w_c;
    case - : w_n := w_n + q * w_c / 2;
  if the n's activity is not null
    Try to flip l's weight; Compute the new net input
    switch(Comp(new net input, old net input))
       case 0 : if (Small(old net input)) r_l := r_l + r_c;
                w_l := w_l + w_c;
       case + : r_l := r_l + w_c;
       case - : w_l := w_l + q * w_c ;
  else
    try to both flip l's weight and modify the n's activity
    Compute the new net input
    switch(Comp(new net input, old net input))
      case 0 : if(Small(old net input)) r_n := r_n + q * r_c;
                                        r_l := r_l + q * r_c;
               w_n := w_n + q * w_c; w_l := w_l + q * w_c;
      case + : r_n := r_n + q * w_c; r_l := r_l + q * w_c;
      case - : w_n := w_n + q * q * w_c; w_l := w_l + q * q * w_c;
```

Figure 7.1: The learning algorithm. The subscript c refers to the currently processed unit, the subscript n refers to a neighbor of that unit and the subscript l refers to a link. Function `Comp` returns 0 if the net input change of sign, else it returns the sign of the difference between the absolute value of its first and second argument. Function `Small` returns 1 if its argument is 0 or 1, else it returns 0.

We now describe the learning algorithm: At each epoch, all the patterns to be learn are forwarded through the network. After each pattern is passed through, the value of two variables are computed for each neuron and for each link: r and w. The letter r stands for right and w stands for wrong. Let o be an output unit, if the activity of o is the desired output, r_o is set to 1 and w_o is set to 0, otherwise, r_o is set to 0 and w_o is set to 1. For a hidden unit h the variables r_h and w_h are computed so as to hint about the correctness of h's computed activity. If r_h is high, a modification of h's activity is likely to decrease the performance of the net on the currently processed pattern. If w_h is high, a modification of h's activity is likely to increase the performance of the net on the currently processed pattern. Similarly, for each link l the variables r_l and w_l are computed in order to hint about the correctness of l's current weight.

Computation of all these variables are performed by processing the neurons one by one, backwards, starting from the output units as in backpropagation. The algorithm sketched in figure 7.1 shows how to process one neuron. Let c be the currently processed neuron. There is three possible trials to modify c's activity:

- t_1 Flip an incoming weight on a link l.

- t_2 Modify the activity of an incoming neighbor n.
- t_3 Both at the same time.

The learning algorithm successively tries t_1, t_2 and t_3. For each trial t_i it analyses the effect of the modification on the neuron c. Suppose that the effect is desirable. It increases w_n if t_i has modified n's activity and w_l if t_i has flipped l's weight. Now, suppose that the effect is bad. It increases r_n if t_i has modified n's activity and r_l if t_i has flipped l's weight. There are three possible effects:

- The net input of c has changed its sign (0 has the sign -). Thus c's activity changes. If $w_c > 0$ we wanted c's activity change with a strength w_c. We register the fact that the trial t_i is worth with a strength w_c. If t_i has modified n's activity we add w_c to w_n, if t_i has flipped l's weight we add w_c to w_l.

  If $r_c > 0$ we did not wanted c's activity to change with a strength r_c. We want to register the fact that the trial t_i is bad with a strength r_c. We add r_c to r_n or to r_l depending on t_i, but this time only if c had the poorest stability, i.e, its net input, before the trial t_i was 0 or 1. If the net input was -1 or 2, the other possible cases, then the effect of t_i can be always counter-balanced by setting n's activity to 0.

- The net input increases in absolute value. If we wanted c's activity to change this effect is bad because the change will be harder. Thus we add w_c to r_n or r_l, depending on t_i.

- The net input decreases in absolute value. If we wanted c's activity change, the effect is desirable since the change will be easier to do. But we are less satisfied than in the first case where the change was completely done, so instead of adding w_c, we add $w_c * q * \delta/2$ to w_n or w_l, where q is a parameter lying between 0 and 1, strictly smaller than 1, and δ is the decrease in the absolute value of the net input. δ depends on t_i, it is 2 if one has made a weight flip, and 1 otherwise. We involve δ to register the fact that the progress of the net input of c toward the correct value by 2 is better than if it is only by 1.

The trial t_3 allows to register an information that is relevant only if we make two weight flips, one flip to change the weight of the incoming link l, another flip upstream the incoming neuron n to change the activity of n. On the other hand, the information computed during t_1 and t_2 is relevant for one weight change. Since the information computed during trial t_3 is less precise, we multiply the backpropagated quantities by q.

Once all the patterns have been processed, the w_l and r_l corresponding to each pattern are summed. We compute a fitness f_l for each link equal to

$$f_l = r * r_l - w * \log(1 + w_l) + d * d_l \tag{7.1}$$

The positive coefficients r, w, d parametrize the learning. In all the experiments, we get some good results using the same set of coefficients. The variable d_l refers to the depth of the link l which is the length of the minimum path from this link to an output unit. The two links in the whole network that have the lower fitness are selected. One of these two links is chosen for flipping. The link with the poorest fitness is flipped with a probability p higher than 0.5. The performance of the neural net with the flipped weight is computed. This performance is the total number of correct outputs divided by the number of patterns divided by the number of output units. If the performance increases, the weight change is accepted, if not, it is still accepted with a probability $\exp(-\Delta/T)$. T is a temperature, Δ is the decay in the performance. From one epoch to the other

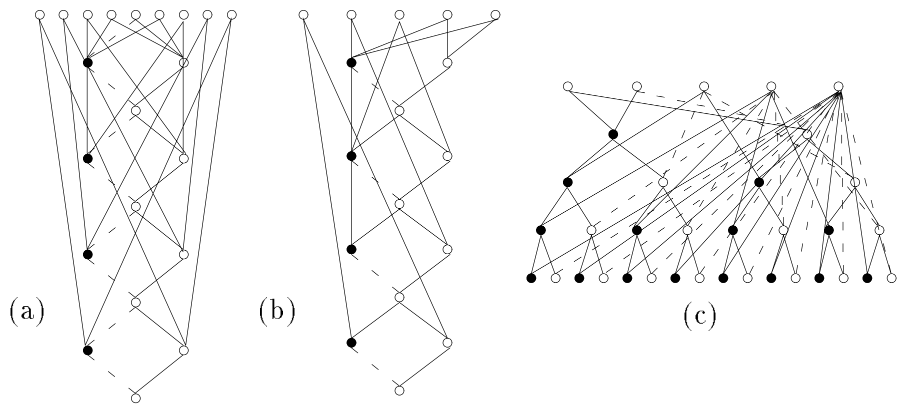

Figure 7.2: Trained neural net for (a) the symmetry, (b) the parity, (c) the decoder. Circles represent neurons. If the threshold is 0, the circle is empty, if it is one the circle is filled with black. A continuous line indicates a weight 1 and a dashed line a weight -1

T is decreased, in order to do simulated annealing. If the change is accepted, then the link that had been selected for change is frozen for e epochs where $e = e_1 + e_2 * d_l$. e_1 and e_2 are positive parameters. Due to the term $e_2 * d_l$, the link near the output units will be frozen a longer time. This counter balanced the fact that they are more likely to be flipped due to the term $d * d_l$ in the expression of the fitness. Thus the algorithm starts by modifying weights of units near the output units, which is better because the information is more relevant there. It then carries on by modifying weights of deep units.

7.3 Simulation with the basic Switch learning algorithm

We now describe three sets of experiments during which we try to learn respectively the symmetry, the parity and the decoder. For each target function we consider a Cellular Encoding (CE) found by hand that produces a family of neural networks with correct architectures and thresholds. We tested the switch learning on the first four networks of the family. These networks are a relevant benchmark, since the switch learning algorithm is designed to be combined with CE. We represent the fourth neural network with the correct weight in figure 7.3. We use two modes for the weight initialization. In the first mode we set all the weights to 1, in the other mode we set the weights with a random value ± 1.

We want to show the influence of the parameter q. If q is set to 0 the algorithm uses only trials t_1 and t_2. It computes the information relevant to a single weight flip. If q is strictly positive, the algorithm computes an information relevant to two weight flips or more. The fitness of a link indicates whether or not the neural net will increase its performance after the flip of this link, and the flip of another particular link. For each set of experiments, we tried the value $q = 0$ and $q = 0.2$. The other parameters were: $r = 5, w = 5, d = 1, e_1 = 2, e_2 = 1, p = 0.8$. The initial temperature is set to 1 and we divide it by 0.01 after each epoch. This parameter set works for the three experiments considered. We run 32 trials for each experiments, and compute the average and the standard deviation SD of the number of epoch needed to reach the solution. The standard deviation of the average can be obtained by dividing SD by $\sqrt{32}$. We want the learning to be fast, and we do not require 100% of success. We therefore allow the learning to run for only 40 epochs.

In table 7.1, we report in this order, the value of q, the mode of weight initialization, the network number which is L, the number of successful trials, the average number of epochs of the successful trials and the standard deviation of the number of epochs.

q	0.00								0.02							
weights	one				random				one				random			
L	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
target	symmetry															
success	32	27	30	22	31	18	6	2	32	32	31	28	32	30	14	12
epoch	2.8	4.4	7.4	13	11	20	21	8	4.3	5.8	11	12	9.4	18	30	30
SD	2.2	1.6	4.2	5.5	7.1	10	9.6	12	5.5	4.2	7.1	7.2	8.6	8.6	6.3	6.9
target	parity															
success	32	32	32	32	30	18	8	2	32	32	32	32	32	32	23	18
epoch	2	2.9	4.4	6.5	6.2	16	17	2	1.7	2.4	3.2	4.7	6.8	13	25	21
SD	3.5	3	3	6	5.6	11	9.2	5.5	2.9	2.3	0.7	3.5	8.4	7.7	9.9	9.0
target	decoder															
success	32	32	32	32	32	32	13	0	32	32	32	32	32	28	15	3
epoch	1	3	7	15	3.9	5.5	4.4	-	1.6	3	7	15	3.6	7.2	14.2	11.5
SD	0	0	0	0	5.1	2.4	8.3	-	2.1	0	0.3	0.3	5.4	4.4	6.8	12.3

Table 7.1: Experimental results with the basic switch algorithm

In the first set of experiments, we try to learn the symmetry function. The network N_L that computes the symmetry of $2L + 1$ inputs has 8*L connections, 2*L layers, 2*L weights must be flipped by the learning, if all the initial weights are 1. The symmetry of $2 * L + 1$ bits input returns 1 if and only if the input is symmetric and the middle bit is 1. The next function studied is the parity. The network N_L has $L + 1$ input units, $2 * L$ layers, $6 * L$ connections, and only L weight to flip if all the weights are initialized with 1. The parity returns the sum of the inputs modulo 2. With the decoder function, N_L has $L+1$ input unit, 2^L output units, L layer, $2^{L+2}-4$ connections, $2^L - 1$ weights have to be flipped. If the first input is 0, the decoder returns 0, else the output unit which number is encoded on the last L input units must be set to 1, and all the others to 0. The result of the fourth decoder neural network with random weight initialization are very poor, This is probably because the network has 60 weights. The learning must flip on average 30 weights which is much more than in the other case.

The experiments show that starting with initial weights of 1 help the learning. When there are a lot of -1 weights, the neuron's activities have a greater probability of being 0. When too many activities are 0, the learning do not have useful information. The experiment also show that setting q to 0.02 enhance the performance, in the case of random weight initialization. In this case, the extra information brought by setting $q = 0.02$ is more precious since little information is available.

We have presented a learning algorithm which aim is to refine neural networks produced by the GA using CE. This learning suits the particular class of neural nets generated by CE. These neural nets have a small number of connections. During one epoch, the learning guesses the best weight to flip, and flips it. It produces ± 1 weights therefore backcoding is possible. The guessing is more difficult to make if there are redundant connections, also if there are more connections, there will be more weights to flip, and therefore more epochs. The neural nets generated with CE have a great number of weights 1, and the learning works better on such networks. They have a lot of layers (up to 8 in our experiment) and the learning can cope with that. Moreover the learning fulfils some special requirements imposed by the GA: It must be quick and need not be 100% successful.

Experiments show that a success rate above 50% is achieved with an average number of epoch around 20, on networks having up to 12 hidden units.

7.4 Extension of the Switch learning Algorithm

In the previous sections, we describe a learning that fits networks generated by the genetic algorithm using cellular encoding. This learning algorithm looks for boolean weights (± 1). Choosing weights within $\{-1, 1\}$ is very restrictive. In this section, we extend this learning so as to choose weights in $\{-1, 0, 1\}$. Setting a weight to 0 amounts to prune the link. So backcoding is still possible.

The extended switch learning algorithm is similar to the basic one. At each epoch, all the patterns to be learned are forwarded through the network. After each pattern is passed through, we update two positive variables for each neuron n: r_n and w_n and six positive variables for each link l: $r_l[v]$ and $w_l[v]$, $v \in \{-1, 0, 1\}$ In fact, we do not consider $r_l[v_0]$ and $w_l[v_0]$, where v_0 is the value of the current weight. The letter r stands for right and w stands for wrong. Let o be an output unit, if the activity of o is the desired output, r_o is set to 1 and w_o is set to 0, otherwise, r_o is set to 0 and w_o is set to 1. For a hidden unit h the variables r_h and w_h are computed so as to give an hint about the correctness of h's computed activity. If r_h is high, the activity of h is right, a modification of h's activity is likely to decrease the performance of the net on the currently processed pattern. If w_h is high, the activity of h could be wrong, a modification of h's activity is likely to increase the performance of the net. Similarly, for each link l, if $r_l[v]$ (resp. $w_l[v]$) is high, setting l's weight to v is likely to decrease (resp. increase) the performance of the net on the currently processed pattern.

In order to compute the value of all these variables we process the neurons one by one, backwards, starting from the output-units, as in back-propagation. The algorithm is similar to the algorithm used in the ± 1 basic switch algorithm. It is sketched in Appendix C.11 and it shows how to process one neuron. Let c be the currently processed neuron. There are three possible trials to modify c's activity: We can either change a weight on a link l that fans into c (trial t_1) or modify the activity of a neighbor n that fans into c (trial t_2) or else do trial t_1 and t_2 at the same time (trial t_3). For each neighbor n that fans into c through link l, the learning algorithm successively tries t_1, and one among t_2 or t_3 depending on the activity of n. For each trial t_i it analyses the effect of the modification on the neuron c. If the effect is desirable, it increases w_n if t_i has modified n'activity and $w_l[v]$ if t_i has set l's weight to v. If the effect is bad, it increases r_n if t_i has modified n's activity and $r_l[v]$ if t_i has set l's weight to v. There are three kind of effects, following the change of the net input of c which is the weighted sum of the neuron's activity that fans into c, minus c's threshold.

- e_1: The net input of c changes its sign (0 has the sign -). Thus c's activity changes. If $w_c > 0$ we wanted c's activity change with a strength w_c. We register the fact that the trial t_i is worth with a strength w_c. We add w_c to w_n or $w_l[v]$ depending on t_i. If $r_c > 0$ we did not want c's activity to change with a strength r_c. We register the fact that t_i is bad with a strength r_c. We add r_c to r_n or to $r_l[v]$ depending on t_i.

- e_2: The net input increases in absolute value. If we wanted c's activity to change, this effect is bad because the change will be harder. Thus we add w_c to r_n or $r_l[v]$, depending on t_i.

- e_3: The net input decreases in absolute value. If we wanted c's activity change, the effect is desirable since the change will be easier. We add w_c to w_n or $w_l[v]$ depending on t_i. If we did not want c's activity to change the effect is bad because c's activity is less stable and will be more easily changed. We add r_c to r_n or to $r_l[v]$ depending on t_i.

Until now, the Switch Algorithm looks like a standard back-propagation. However there is a difference. The quantities r_c and w_c that are back propagated, are multiplied by a coefficient which is used to rank the relative importance of the corresponding change in a weight value or an activity value. The basic switch learning algorithm also uses coefficients, but not as many. Moreover the extended switch learning consider more cases. The coefficients are produced using 6 parameters p_i, $i = 1, \ldots, 6; 0 < p_i < 1$. Each of these parameter, corresponds to a particular heuristic. Parameter p_1 promotes the change of the activity of a neuron having a small stability. This kind of strategy has already been used in [8]. It has theoretical and experimental foundations. Parameters p_2, p_3 and p_4 rank the importance of the effect e_1, e_2, and the two cases of e_3. We prefer to try to learn new patterns rather than consolidate old patterns. This heuristic can be implemented by choosing $p_3 >> p_4$. An input unit's activity is fixed. Therefore we can better evaluate the effect of a weight change on a link that fans out an input unit. Parameter p_5 is used to register this fact. The trial t_3 allows to register an information that is relevant only if we make at least two weight changes, one to change the weight of the link l that fans in c, another changes a weight upstream the neuron n connected by l in order to modify n's activity. On the other hand, the information computed using t_1 or t_2 is relevant for one weight change. Since the information computed during trial t_3 is less precise, we multiply the back propagated quantities by p_6 in t_3. The effect e_3 is more important if the change in the net input is 2, than if it is 1. Therefore, we multiply the backpropagated quantities by $\delta/2$ where δ is the absolute value of the difference between the net input before the trial and the net input after the trial. The algorithm in Appendix C.11 can be implemented with 8 multiplications, 7 additions and 4 conditional branching for processing one link, by computing the coefficients in advance, for each possible c's net input, l's weight and n's activity.

Once all the patterns have been processed, the $w_l[v]$ and $r_l[v]$ corresponding to each pattern are summed. A fitness $f_l[v]$ is computed for each link $l, v \in \{-1, 0, 1\} \setminus \{v_0\}$, where v_0 refers to the current weight. We use the formula: $f_l[v] = w * w_l[v] - r * r_l[v] - d * d_l$. The positive coefficients r, w and d parametrize the learning. The fitness $f_l[v]$ measures how interesting would be to set l's weight to the value v. The variable d_l refers to the depth of the link l which is the length of the maximum path from this link to an output unit. The fitness of a link f_l is the maximum of $f_l[v]$, The two links in the whole network that have the higher fitness are selected. One of these two links is chosen for a weight change. The one with the highest fitness is chosen with a probability p higher than 0.5. The weight of the selected link is set to v where $f_l[v]$ is the maximum fitness. After the weight change, the performance of the neural net with the modified weight is computed. If the performance increases, we accept the weight change, if not, we still accept it with a probability $\exp(-\Delta/T)$. T is a temperature, Δ is the decrease in the performance. From one epoch to the other T is decreased, in order to do simulated annealing. If the weight change is accepted, then the weight of the link that has been selected for change is frozen for e epochs where $e = e_1 + e_2 * e_3^{d_l}$, e_1, e_2 e_3 are positive parameters. Due to the term $e_2 * e_3^{d_l}$ in this expression, the links near the output units will be frozen a longer time. This counter balanced the fact that these links are more likely to be selected, due to the term $-d * d_l$ that appears in the computation of the fitness f_l.

7.5 Simulation with the extended Switch learning algorithm

We now describe two sets of experiments during which we try to learn 2 boolean functions: the parity and the symmetry. For each target function, we consider a genetic code found by hand that

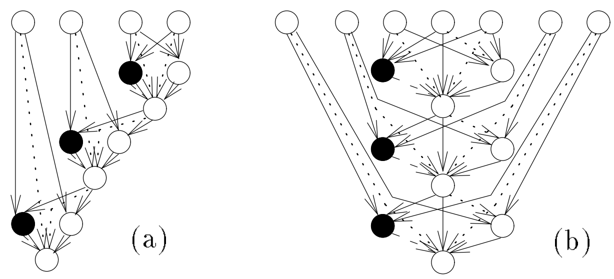

Figure 7.3: The third learned network for the parity (a) and the symmetry (b). Circles represent neurons. If the threshold is 0, the circle is empty, if it is one the circle is filled with black. A continuous line indicates a weight 1, a dashed line a weight -1 and a dotted line a null weight.

produces a family of neural networks such that some links must be pruned by learning in order to find a solution. We tested the Switch algorithm on the first three neural networks of the family. Networks developed with cellular encoding are a relevant benchmark, since the learning algorithm is designed to be combined with cellular encoding. We represent the third network of the family, with a set of correct weights found by the learning in fig 7.3. We use two modes for the weight initialization: In the first mode we set all the initial weights to 1, in the other mode, we set the weights with a random value ± 1. For the symmetry we set the parameters to $p_1 = 0.3$, $p_2 = 0.5$, $p_3 = 0.04$, $p_4 = 0.006$, $p_5 = 0.17$, $p_6 == 0.02$, $r = 5$, $w = 7$, $p = 0.8$, $d = 0.5$, $e_1 = 2$, $e_2 = 1$, $e_3 = 0.7$. We keep the same parameters for the parity except for $w = 6$, $d = 2$, $p_1 = 0.7$. The initial temperature is set to 1 and we divide it by 1.01 after each epoch. We run 32 trials for each experiment, and compute the average and standard deviation. Since the problem is more difficult than the one studied with the basic switch learning (there are more weights, and three possible values per weights), we allow up to 100 epochs before stating success or failure. In the following table the rows report in this order, the target application, the mode of weight initialization, the network number which is L, the number of successful trials, the average number of epochs of the successful trials and the standard deviation of the number of epochs.

target	parity						symmetry					
weights	one			random			one			random		
L	1	2	3	1	2	3	1	2	3	1	2	3
success	32	32	32	32	21	7	32	32	31	32	15	8
epoch	3	6	17	8	22	24	3	7	16	16	41	34
SD	1	1	7	6	24	22	0	4	12	14	22	26

In the first set of experiments, the target is the parity function. The parity of $l+1$ binary inputs returns the sum of its inputs modulo 2. We used a code found by hand that develops a network family (N_l) such that (N_l) has $l+1$ input units, $3*l$ hidden units, $8*l$ connections, $2*l$ layers, $3*L$ weights must be modified by the learning, if all the weights are initialized with 1. In the next set of experiments we try to learn the symmetry function. The symmetry of $2l+1$ binary inputs returns 1 if and only if the input is symmetric and the middle bit is 1. We choose a code that develops a network family (N_l) with the following features: Network (N_l) has $2l+1$ input units, $3*l$ hidden units, $11*l$ connections, $2*l$ layers. $4*L$ weights must be modified by the learning, if all the weights are initialized with 1.

LEARNING MODE	TIME	EVALUATIONS	SUCCESS-RATE	NUMBER OF TRIALS
basic GA	110	11500	0.14	100
Hebbian Learning only on the output unit				
Baldwin	260	17700	0.07	100
Developmental	21	1800	0.64	76
Lamarkian	13	1300	0.79	63
Switch Learning on all hidden layers				
Baldwin	180	6500	0.08	100
Developmental	11	520	0.84	59
Lamarkian	10	610	0.86	58

Table 7.2: The parity experiment

Experiment shows that a success rate above 20% is achieved with an average number of epochs under 40, on networks having up to 8 hidden units. This is not as good as the basic switch learning. But the complexity of the problem studied is also more important. Supose we start with all the weights initialized with one. There were three times as much weights to modify in the case of the parity, and twice as much in the case of the symmetry. Moreover, there are two choices instead of one for the new value of the weights. As was the case with the basic Switch learning, the experiments also show that starting with initial weights of 1 helps the Switch Algorithm.

7.6 Simulation with the Genetic Algorithm

In this set of experiments, we combine the Genetic Algorithm with the basic Switch learning We use the basic Switch learning algorithm that searches ± 1 weights, because the experiments of the preceding sections indicate that the extended version where the weights are chosen in$\{0, 1, -1\}$ is less efficient on the particular target problems that we want to solve. We used the parallel GA described in chapter 8 and made it run on four nodes of an IPSC860, a MIMD parallel Machine. We compared results obtained using Hebbian Learning on the output unit and Basic Switch Learning on all units. We consider two problems to solve: the parity and symmetry problem of arbitrary large size. The 3 possible combinations of learning and evolution described in the preceding chapter are investigated. they are: Baldwin Learning, Developmental Learning, and Lamarkian Learning. The first table reports the parity experiments. The population size is now fixed, it is 256, the GA runs 20 seconds before stating success of failure. The time for the failures is considered in the average time we give. The second table reports the symmetry experiments. Here the population is 1024, the GA runs 300 seconds before stating success or failure.

These experiments clearly show that Switch learning allows to get a better speed-up than hebbian learning, especially for the symmetry problem. This is not surprising, since learning only on the output unit exploits a special property of the parity problem; recall that for the network for the parity problem of 2 inputs, the only weights that need to be negative are weights feeding the output unit. In the cellular encoding weights are positive by default; therefore, encodings with all positive weights are more compact than encodings for equivalent networks that have negative weights. On the other hand, Symmetry is a problem for which the learning should modify weights of hidden units. Switch learning allows to learn weights on links feeding hidden units, and the hebbian learning used does not. This is why the interest of switch learning shows up on the

LEARNING MODE	TIME	EVALUATIONS	SUCCESS-RATE	NUMBER OF TRIALS
basic GA	790	44000	0.27	71
Hebbian Learning only on the output unit				
Baldwin	960	48000	0.15	84
Developmental	360	20000	.57	14
Lamarkian	420	24000	0.5	40
Switch Learning on all hidden layers				
Baldwin	2280	45000	.12	170
Developmental	62	1400	1	23
Lamarkian	98	2850	0.74	27

Table 7.3: The symmetry experiment

symmetry problem. On this problem, Developmental learning does better that Lamarkian learning

7.7 Conclusion

The "Switch Algorithm" has been designed to refine neural networks produced by the Genetic Algorithm, with Cellular Encoding. During one epoch, the Switch algorithm guesses the best weight to change, and also the best value into which to change it. This value can be ± 1 in the basic switch algorithm $-1, 0$ or 1 in the extended case. This learning suits the particular class of neural nets that are generated by the GA. These neural nets are made of units having a small fan in, and therefore, the number of connections is also small. They have a great number of weights 1, and many layers. Moreover, the Switch Algorithm is quick and the computer time spent into learning is comparable to the time spent by the GA alone, for the genetic search and the development of neural networks. We did two kinds of experiments to measure the efficiency of switch learning. First we measure the speed of convergence on a fixed neural net with correct architecture and biases. Second, we measure the speed-up brought to the GA. The first kind of experiments with the basic switch learning shows that a success rate above 50% is achieved with an average number of epochs around 20, on networks having up to 12 hidden units. When combined with the GA, with the symmetry problem, the basic Switch learning allows to get a speed-up of 13, whereas the Hebbian learning on the output unit brings a speed-up of only 1.5.

Chapter 8

The mixed parallel GA

8.1 State of the art of Parallel Genetic Algorithms

The Genetic Algorithm is a search technique based on the principles of population genetics. It has been shortly describes, section 1.2, in the introduction. A GA handles a population of solutions to a given problem. In order to build a new solution, the GA selects two solutions in its population (mating) recombines them to build an offspring, and finally mutates the offspring. This iterative process of selection, recombination and mutation makes the population of solutions evolve towards better solutions. The process is stopped when a sufficiently good solution is found. We refer to [11] for a book on GAs.

The basis of a parallel implementation of a GA on a multiprocessor system is to divide the whole population into subpopulations and to allocate one subpopulation per processor. The processors send each other their best solutions. These communications take place with respect to a **spatial structure** of the population. Different models of parallel GAs have been investigated. There are based on different **spatial structures**. Müehlenbein classifies 3 models in [24]. In the island model, the population is subdivided into a set of randomly distributed islands among which migration is random. In the stepping-stone model, migration takes place between neighboring islands only. The isolating-by-distance model treats the case of continuous distributions. Typically, in this model, individuals are placed on a 2-D grid, and mate in their neighborhood.

8.2 The mixed parallel GA

In this chapter we introduce a new model called "the mixed model", which combines the stepping-stone model and the isolating-by-distance model. Thus the advantages of both models are combined.

- Local mating allows to achieve a high degree of inbreeding between mates ([4]). The degree of inbreeding is a measure of how similar the mates are on average. If the mates are very different, their combination is likely to produce unfit offsprings. Thus a high degree of inbreeding favors the creation of interesting individuals.

- Isolated islands help to maintain genetic diversity ([25]). Individuals on an island are relatively isolated from individuals on another island. Therefore the sub-population of each island is exploring a different part of the input-space. It avoids premature convergence and allows a more thorough search of the input space.

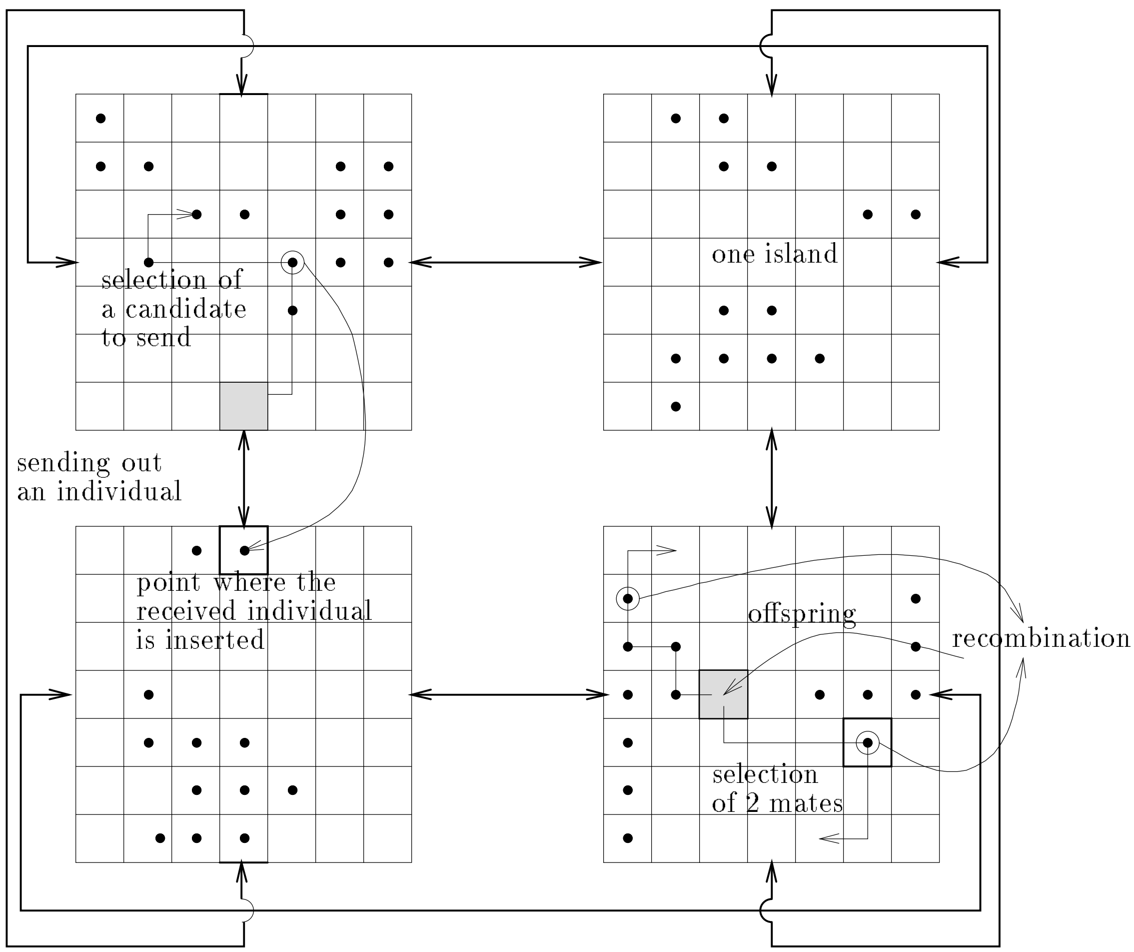

Figure 8.1: Spatial structure of the population, for the mixed parallel GA: A 2-D torus of 2-D grid. each small square represents a site, a dot in a square represents an individual occupying the site, a cercle around the dot indicates that this individual is the best of a random walk. Grey squares represent a site that has been randomly chosen to initiate a random walk, thick squares are site where an individual is to be inserted

In the mixed parallel GA, individuals are distributed on islands. The islands form a 2-D torus. Each island is mapped on one processor of a MIMD machine. A processor can send individuals only to the four processors that store the four neighbor islands. Inside one island the individuals are arranged on a 2-D grid. The whole spatial structure is a 2D torus of 2-D grids represented fig 8.1.

Not all the sites of a 2-D grid are occupied. The density of population is kept around 0.5. It means: a given site is occupied with a probability 0.5. The mating is done as follows: A site s is randomly chosen on the grid. From this site, two successive random walks are performed. The two best individuals found during these 2 random walks are mated. A verification is done to ensure that the two mates are different. The offspring is placed on site s. If the site was already occupied by individual i, i is placed on a randomly chosen neighboring site s'. If then s' is occupied by i', i'

is placed on a randomly chosen neighboring site s'', and so on until an empty site is found. This scheme simulates the isolating-by-distance model sequentially on one processor.

Tanese [30] introduces two migration parameters: the migration interval (the number of generations between consecutive migrations) and the migration rate (the number of individuals selected for migration). This approach is usually followed. We take an original approach that requires only one parameter. Our GA is steady state. Each time a new individual is to be created, it can be built either through the recombination of two other individuals, or by mean of an exchange. The exchange is chosen with a probability called the exchange rate.

When a processor exchanges an individual, a site s is selected on a border of the 2-D grid, a random walk starts from this site, the best individual found is sent to the processor next to the border. It is sent with the coordinate of s. When it is received, these coordinates are read, and the individual is placed exactly on the site opposite to s. Thus, the exchange preserves the overall 2-D topological structure.

8.3 A super-linear speed-up

We apply the GA to find a boolean neural net (with weight ± 1) that computes the parity of 51 inputs. We encode neural nets with the Cellular Encoding. This encoding allows to develop a family of neural networks, where the L_{th} network computes a regular boolean function of size L like the parity with $L+1$ inputs. Müehlenbein [25] promotes the use of individual hill-climbing. Each individual generated by the GA should try to enhance itself with hill-climbing. In the case of neural network optimization, a learning procedure can be used as a hill-climber. The basic switch learning was used. This learning is applied for each chromosome (solution) produced by the GA, on the first neural network of the family. The learned information is then coded back on the chromosome and can be used to develop the other networks of the family. This method called developmental learning has been studied in chapter 6.

The MIMD parallel machine used is an iPSC860, with up to 64 processors. One node of this machine makes 40 MIPS which is roughly 1.5 SPARC-2. The communications between the processors are asynchronous, and are overlapped by the computations. So the communication cost is null! This theoretical prediction is confirmed by experimental measures. If the GA failed to find a solution after 20 seconds, it was restarted. The time to find a solution includes the time spent for the failure. The optimal size of the population changes with the number of processors used. Therefore, in the study of the speed-up, we have considered different population size. Figure 8.2 shows that the optimal population size depends on the number of processors. Some authors claim a super-linear speed-up for their parallel GA, but their results are biased because they choose a population size that favors a great number of processors. It is well known that the performance of the GA and the parallel GA depends critically on the population size. Therefore we believe that a computation of the speed-up which do not try to optimize the population size for a given number of processors is of little interest.

Figure 8.3 shows a super-linear speed-up until 16 processors. This is possible since the parallel GA is not the same algorithm as the basic GA. The speed-up drops after 16 processors not because of the communication cost which remains null, but because roughly, 128 individuals are enough to solve the target problem; and the Mixed PGA works well for sub-populations of at least 8 individuals, With a more difficult problem, a greater population size would be needed to correctly sample the search space, and a super-linear speed-up for an even greater number of processors could be achieved.

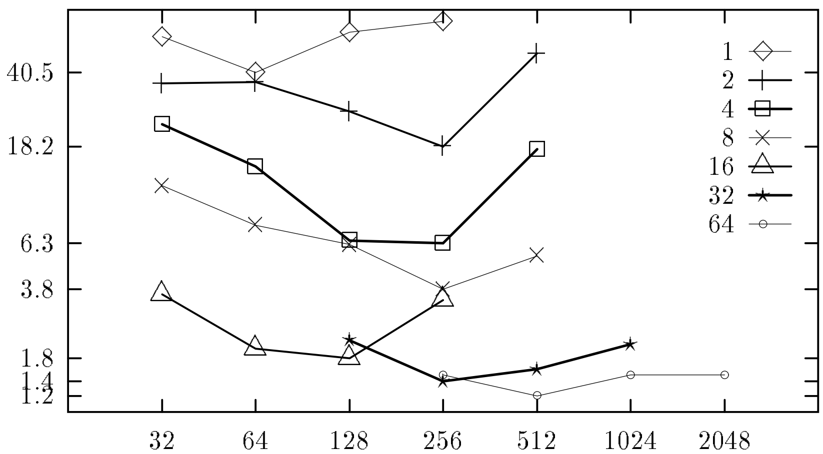

Figure 8.2: The horizontal axis indicates the total population size (summed on all the processors) which is always a power of 2. The vertical axis indicates the time in seconds, taken by the Mixed Parallel GA, to find a network for the parity of 51 inputs. It is an average over 10 experiments, depending on the initial population. Tics on the vertical axis indicate the best time for each number of processors

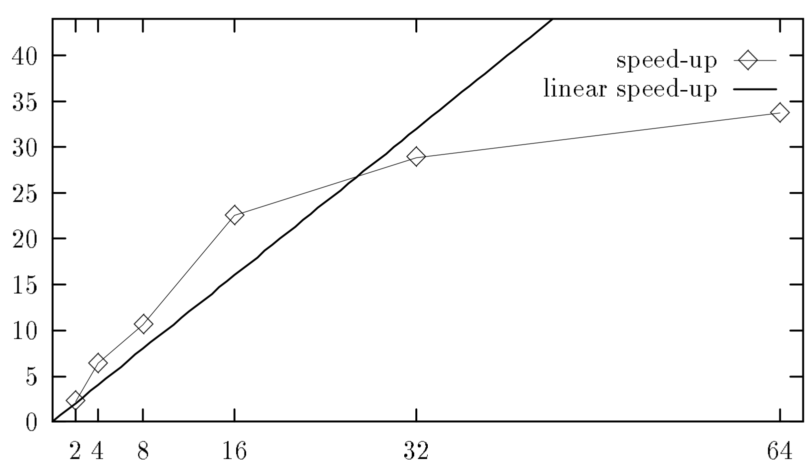

Figure 8.3: The horizontal axis indicates the number of nodes, the vertical axix shows the speed-up

Chapter 9

Modularity

This chapter describes how to incorporate modularity into Cellular Encoding. We show how to add modularity to the basic Cellular Encoding, A Genetic Algorithm that encompasses all the enhancements brought in the previous chapters, is used to find part of a modular code that yields both architecture and ± 1 weights specifying a particular neural network for solving the decoder boolean function of 10 inputs, 1024 outputs. This result suggests that the GA can exploit modularity in order to find architectures within a more complex range.

9.1 Adding modularity to Cellular Encoding

In section 3.4 we have shown that using the program symbol `JMP`, we could obtain a cellular encoding that checks property of modularity. We have tested this operator without much success. We therefore choose another way to implement modularity.

A cellular code is now extended to an ordered list of labeled trees, instead of a single tree. Since each tree encodes a grammar it is called a grammar-tree. Each grammar tree encodes a sub neural network that will be included in the final neural network. The GA must automatically define these sub neural networks, and include some copies of them in the final network. Koza in [17], describes how the GA can automatically define functions, and insert one or many calls to these functions. Koza calls that Automatic Function Definition. We shall call that ADSN: Automatic Definition of Sub Networks. Each grammar-tree has its own name which for convenience will be a single lower-case letter, except for the *axiom grammar tree*, whose name is $\#$. At the beginning of the rewriting, the ancestor cell is positioned on the root of the axiom grammar tree. For each grammar-tree $\mathbf{x}$, there exists a program-symbol denoted by $\mathbf{x}$ which includes the sub network encoded by that grammar tree. This operator is similar to the program-symbol `JMP x` defined in section 4.6. The only difference is that we encode the program-symbol on a single letter, and it has no arguments. There are as many macro-operators as there are grammar-trees. This notation is coherent with the use of Genetic Algorithm. When a cell reads the program-symbol $\mathbf{x}$, it positions its reading head on the root of the tree which name is $\mathbf{x}$. If a grammar tree includes itself, the development enters an infinite loop. It amounts to use a program symbol of recursion $\mathbf{R}$ defined in section 2.2 In this particular case, the cell executes the following algorithm before moving its reading head, that counts the number of loops:

```
1- life := life - 1
2- If (life > 0)
      reading-head := root of current tree
```

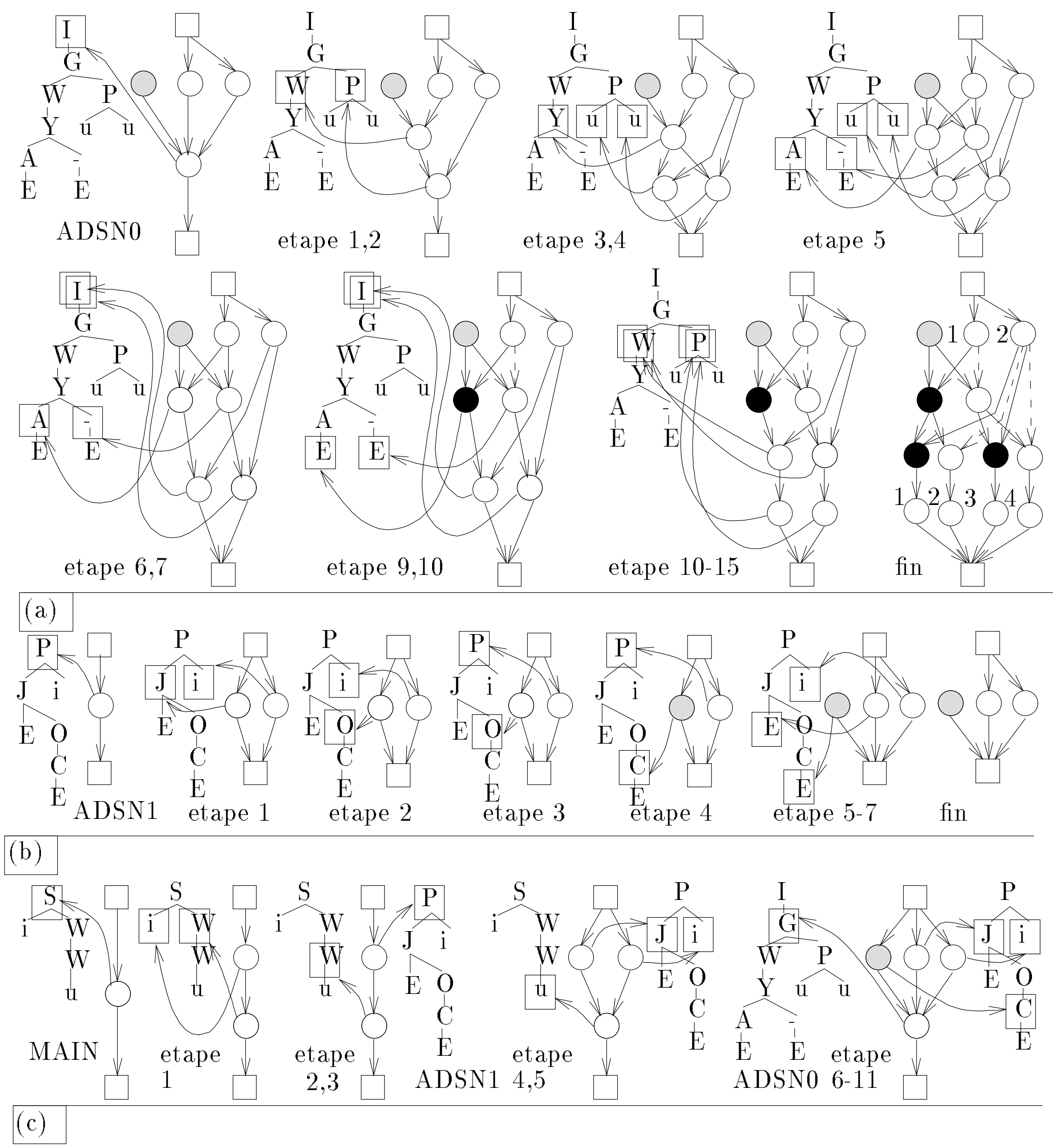

Figure 9.1: The 2-input decoder network. (a) Development of sub-network **u**, starting from an initial network graph already containing subnetwork **i**. (b) Development of sub-network **i**. (c) First 11 steps of the development of the whole 2-input decoder network, at step 11, the network-graph is similar to the initial network-graph of (a), the last steps can therefore be deduced from (a). The graphic representation is as usual, except that a circle filled with gray represents a threshold of -1. We use an abbreviated notation for program symbols. `PAR, SEQ, WAIT, END, INCBIAS, DECBIAS, VAL-,VAL+, INCLR, DECLR, CUT, XSPL, SEP, ADSN1, ADSN0` are replaced by: `P, S, W, E, A, O, -, +, I, D, C, G, Y, u, i,`

```
3- Else loose reading-head
      become a neuron.
```

The variable *life* is a register of the cell. It is initialized with L in the ancestor cell. Thus a grammar develops a family of neural networks parametrized by L. The number L parametrizes the structure of the neural network. We also introduce a branching program-symbol denoted `BLIF2` which labels nodes of arity 2. A cell that reads `BLIF2` executes the following algorithm:

```
1- If (life = L)
      reading-head := right subtree
                      of current node
2- Else
      reading-head := left subtree
                      of current node
```

Figure 9.1 shows a modular cellular code and its development into a network for the decoder. This figure is complex, we refer the reader to chapter 2 for a more simple example of development and an explanation of cellular encoding. We present here a more complex encoding for three purposes. It shows the power of cellular encoding. It presents a complex boolean function, the decoder, for which a modular decomposition is **needed** to encode the corresponding network family. It allows to better understand how the GA can find a solution to the decoder problem.

Figure 9.1 (c) shows that the decoder-network consists of 2 sub-networks sequentially connected. The first one is encoded by `ADSN1` and develops the input units, as well as a unit which constantly outputs 1. It includes itself once, therefore its size is $O(L)$. The second one is encoded by `ADSN0`. It uses one by one the neurons generated by sub-network `ADSN1`. It includes itself twice, therefore its size is $O(2^L)$. The decoder network is developed for $L = 2$ in figure 9.1. The development can be done for an arbitrary large integer L. Moreover, the code is modular according to the definition given in section 3.4. In general, a cellular encoding will always be modular, by using one grammar-tree per sub-network.

We use two new division program-symbols. In a *gradual division* (denoted by `XSPL`) the first child cell inherits the first r input links, where r is the value of the link register, the second child cell inherits the other input links and all the output links; the first child connects to the second with weight 1 (in step 2, 14 and 15 fig. 9.1 (a)). In a *separation* division (denoted by `SEP`), both childs inherit all the input links, the first child inherits the first half of the output links, the second child inherits the second half (step 5 fig. 9.1 (a)). These program symbol `XSPL` and `SEP` and their microcode are listed in appendix C.5.

9.2 Experiments with modularity

We try to find a network family N_L such that N_L computes the decoder of size L. N_L must have L input units and 2^L output units. If the binary code of an integer $i < 2^L$ is clamped on the input units, the i^{th} output must be set to 1, all the others to 0. This problem is more complex than the parity or the symmetry, with respect to Cellular Encoding. In the preceding section, we have presented an encoding for a decoder network that needs 3 grammar trees, whereas parity or symmetry networks can be encoded with a single grammar-tree.

We will use all the enhancement brought in the previous chapters to solve this complex problem: Developmental learning, basic Switch learning, parallel GA on 64 processors of an IPSC860 parallel machine. In the parallel GA, we start to develop neural networks from $N_{L_{min}}$, instead of from N_1,

Objective:	Find the Cellular Encoding of a neural network family that computes the decoder of an arbitrary large size
Terminal set:	`END, ADSN0, ADSN1, CONST1`
Function set:	`PAR, SEQ, WAIT, VAL+, VAL-, INCLR, DECLR, INCBIAS, DECBIAS, CUT BLIFE2, XSPL, SEP`
Fitness cases:	Random patterns of L bit to input in the neural net N_L, the number of fitness cases is determined by the GA
Hits:	Fitness cases for which the neural net produces the right output
Raw fitness: of one neural net	The number of hits of the neural net
Standardized fitness of one neural net	raw fitness divided by the total number of fitness cases. It ranges between 0 and 1
Standardized fitness	The sum of standardized fitness of the developed neural net. It ranges between 0 and 10
Parameters:	$M = 8092 t_m = 0.005, l_0 = 30 l_{max} = 50$
Success Predicate:	The standardized fitness equals 10. That is the 10 first networks are developed and score the maximum number of hits

Table 9.1: Tableau for genetic search of a neural network family for the decoder boolean problem. Instead of the time T, we report M, the size of the population which is fixed in this experiments

using the developing rule of section 5.2. L_{min} varies regularly among the 64 processors between 1 and 3. The processors for which $L_{min} = 3$ directly start to develop N_3 whereas those for which $L_{min} = 1$ start by N_1. There are three different fitness functions, each one represents a different feature of the problem. So each processor develops a population with a view to find the building block corresponding to a particular feature, and the final solution can be found through the recombination of exchanged genomes between processors. Genetic diversity is maintained until a code that develops N_1, N_2, N_3 right is found. But such a code is likely to develop N_L right for $L > 3$, because the recurrence is learned. Hence genetic diversity can be kept until a general solution is found.

The grammar-tree are ordered. Mutation and the generation of random genomes are implemented so as to respect a hierarchy between the grammar-tree of the genome. A grammar-tree **x** cannot include a grammar-tree **y** if **x**<**y**. This prevents looping between grammar-trees.

The fitness of a neural networks is no longer based on information theory, but simply on the number of hits. This is consistent with respect to the Switch learning that tries to maximize the number of hits.

The genome recombined by the GA encompassed two grammar-trees named `ADSN0` and `ADSN1`. The axiom grammar-tree was predefined as `SEQ (ADSN1 )( WAIT ( WAIT( ADSN0)))`. This tree indicates the general structure of the network. It is made of two sub-networks `ADSN1` and `ASDN0`, sequentially connected. The development of `ASDN0` is delayed 2 times, because `ASDN0` needs neurons produced by `ASDN1`. We predefined another grammar-tree named `CONST1` which is `BLIFE2(END )(DECBIAS (VAL- (CUT ( END))))` . It encodes a sub-network that consists of a single unit, which outputs a constant 1, or performs the identity, depending on the value of the life register. Table 9.1 describes the features for the problem of generating a neural network for the decoder. Figures 9.2 shows the genome found by the GA. The corresponding L^{th} neural network solves the decoder of L inputs with $4 * 2^L - 2$ hidden units and $15 * 2^{L-1} - 6$ links. It is clearly a combination of $2^L - 1$ sub-networks. We tested N_L until L=10. Although there is no formal proof, it seems obvious that N_L solves the decoder of L inputs for an arbitrary large L, because the GA learns the recurrence. The GA used was quite powerful. The sub-population size on each processors was fixed, it was 128, the total population size on the 64 processors was 8192. The number of generations averaged over the processors was about 130. So the total number of genomes processed was around 1 000

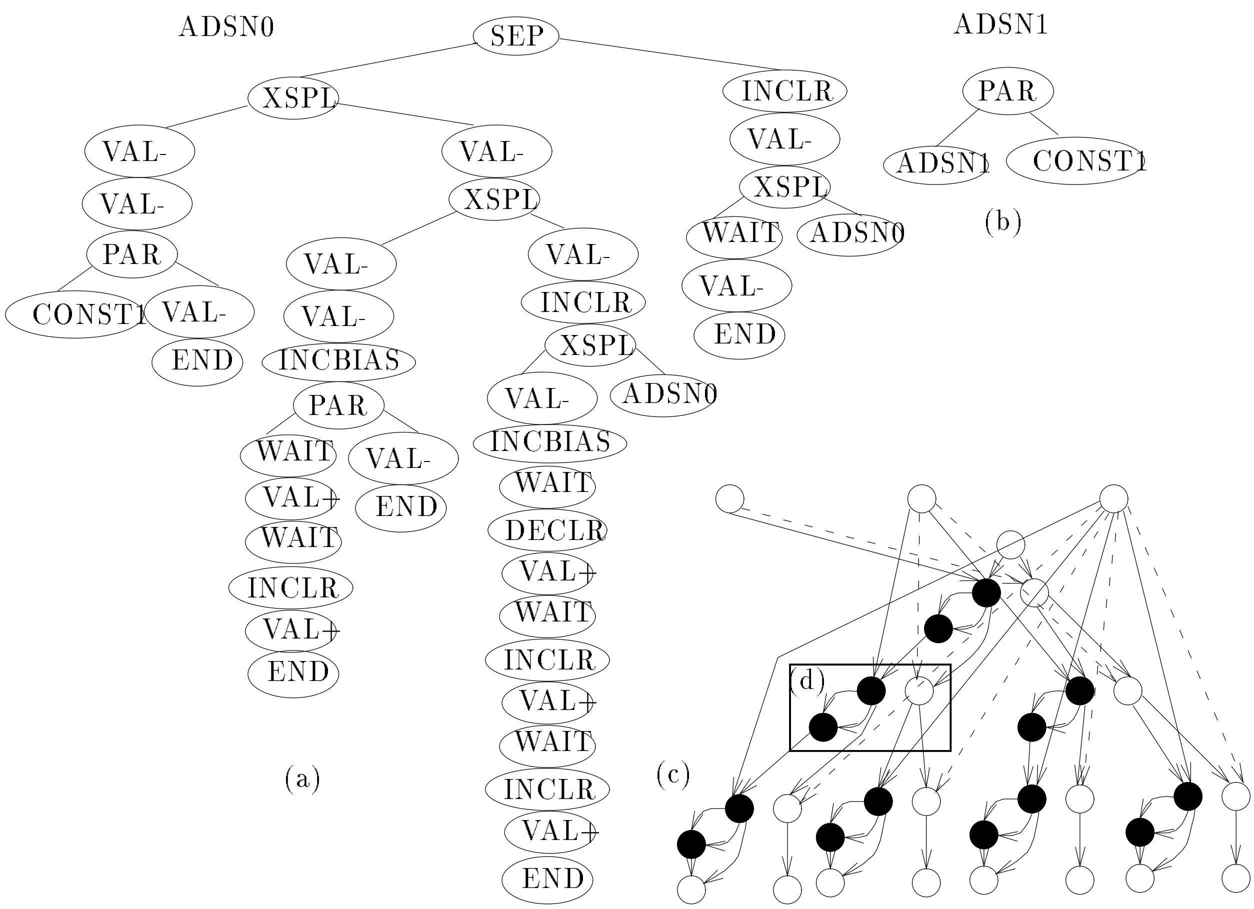

Figure 9.2: (a) The genome found by the GA. (b) the neural net for L=3.

000. The total cpu time taken was one hour, it amounts roughly to four days of sun4, sparc2 workstation.

9.3 Conclusion

In this chapter we make precise how to add modularity to cellular encoding so that the GA can use it. The improvement allows to encode neural networks in a modular fashion. We show how to develop a boolean neural network for the decoder boolean function, using two sub-networks. Experiments report that the GA could simultaneously evolve the two sub-networks and generate a neural network that computes the decoder boolean function of 10 inputs, 1024 outputs. However, the GA had to be helped with two pieces of code provided by hand. The first one describes how to include the sub-networks searched by the GA, the other one encodes a sub-network which can be then used by the GA as a building block. This way of adding information is a general technic, interesting in itself. The decoder is a boolean function which belongs to a more complex class compared to the already studied parity and the symmetry, with respect to Cellular Encoding. The corresponding architecture although regular, is more elaborate. For parity and symmetry, the architecture described in figure 5.9 5.10 are layered, the fan-out is bounded by 2, the complexity is linear. For the decoder network in figure 9.3, links are arranged in a structure of trees, the fan-out is unbounded, the complexity is exponential. Because the decoder was so hard, we had to design an improved GA that efficiently preserves genetic diversity, and includes all the enhancements described in the 3 preceding chapter.

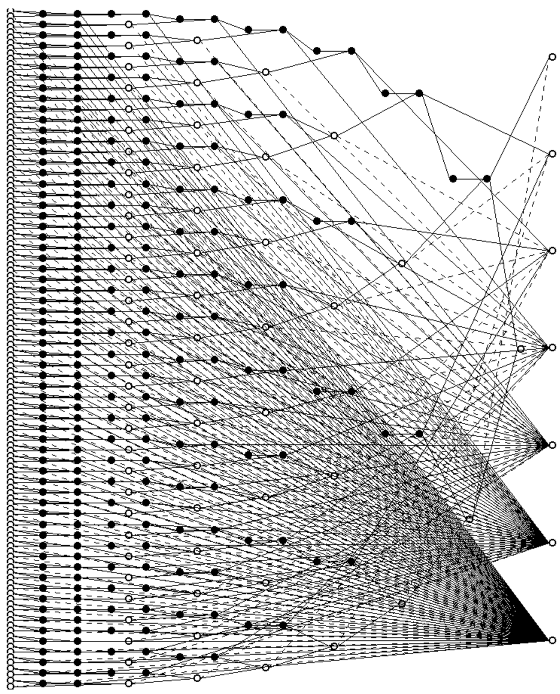

Figure 9.3: The neural network found by the GA for the decoder of 7 input units, 128 output units.

Chapter 10

Conclusion: GA + CE = Genetic Programming of neural networks

Throughout this thesis, we have studied Cellular Encoding which is a method for encoding families of similarly structured boolean neural networks, that can compute scalable boolean functions. The object that is encoded is a parallel graph grammar. The grammar is encoded as a set of trees (called *grammar trees*) instead of a set of rules. Cellular Encoding can describe networks in an elegant and compact way. Moreover, the representation can be readily recombined by the Genetic Algorithm. Various properties of Cellular Encoding have been formalized and proved. A GA has been used to search the space of grammar-trees. The GA can find a Cellular Encoding that yields both architecture and ± 1 weights specifying a particular neural network for solving the parity, symmetry and decoder boolean function . Furthermore, the grammar trees are recursive encodings that generate whole families of networks that compute parity, symmetry or decoder. In this way, once small parity, symmetry or decoder problems are solved, the grammars typically can generate solutions to parity, symmetry and decoder problems of arbitrary large size.

Koza has shown that the tree data structure is an efficient encoding which can be effectively recombined by genetic operators. *Genetic Programming* has been defined by Koza as a way of evolving computer programs with a GA [16]. In Koza's Genetic Programming paradigm the individuals in the population are LISP S-expressions which can be depicted graphically as rooted, point-labeled trees with ordered branches. Since the cellular code has exactly the same structure, the same approach is used for generating and recombing cellular codes with the GA. As a conclusion of our thesis, we would like to show that the genetic search of the space of Cellular Encodings is really a Genetic Programming approach, not only because the genetic representation uses the same tree data structure.

We now introduce this last contribution. First Cellular Encoding can be considered as a low level programming language of neural networks since there exists a compiler JaNNeT (Just an Automatic Neural NEtwork Translator). that inputs a Pascal program and outputs the cellular code of a neural net that computes what the Pascal program

We make another step that allows to put Cellular Encoding and LISP on the same level, and to compare them. The concept of genetic programming language is defined. LISP and Cellular Encoding are two distinct genetic programming languages. A criterion to classify genetic languages with increasing complexities is introduced. Using this criterion, LISP turns out to be more complex than Cellular Encoding. We propose a hierarchy of genetic languages that starts from a very simple micro programming language and increases continuously in complexity to reach a high level language LISP equipped with libraries. We then address the question of which genetic language is better. We show that the combination of Genetic Programming with neural networks using Cellular Encoding is totally different from the one using the encapsulation operator previously proposed by Koza and Rice using LISP [15]. The advantages of Cellular Encoding are put forward. With all the results of Koza, LISP is better when the task involves manipulation of symbols. On the other hand, this is not the case when the task is to synthesize neural networks. A scheme for the evolution of the genetic language itself is also briefly sketched. In this scheme an ideal genetic language appears as a hierarchy of building blocks. The last section summerizes the new paradigm underlying genetic programming of neural networks.

10.1 A hierarchy of Genetic Language

In this section is given a definition of genetic language, a criterion to classify genetic languages, and a list of genetic languages of increasing complexity.

Definition 4 *A genetic programming language $\mathcal{L}_S$ is a set S of symbols of different arities. A genetic program in such a language is an ordered set of trees labeled by symbols, where at each node, the number of child nodes is the arity of the labeling symbol. Syntactic constraints can limit the range of possible genetic programs.*

Koza distinguished between symbols of arity 0 which he calls terminals, and symbols of arity greater than 0 which he calls functions. Here this distinction would be misleading, because a function symbol does not correspond to a real function. The definition of a genetic program gives a theoretical skeleton. Its goal is to capture the most useful property of programming which is modularity. Each tree t represents a piece of code for a particular function f. This code can be called from other trees, using the number associated to t. With Cellular Encoding, the calling of function f amounts to the inclusion of the neural network that computes f. The calling is implemented using a program-symbol called `JMP` defined in section 4.6. This program-symbol has an argument. When a cell reads this program-symbol, it places its reading head on the root of the grammar tree which number is the argument.

Instead of using the tree number to refer to the trees, it is possible to use a pattern matching process like the one used by Koza (chapter on spontaneous self replication). Section 3.4 describes how this can be done. Although this referring system can be interesting from a genetic point of view, the general structure of a genetic program is the same, it can always be decomposed into a set of trees. It is only a different implementation. We consider a special kind of grammar defined as follows:

Definition 5 *A GEN-grammar is a tree-grammar used to rewrite a genetic program written in a language $\mathcal{L}_S$ into a genetic program written in a language $\mathcal{L}_{S'}$. A rule r of the grammar must check the following condition: The left member is a symbol s of S of arity r. The right member is a tree labeled with symbols of S' and special symbols $T_i, i = 1, \ldots, r$. The right member contains one and exactly one occurrence of each special symbol. When s is rewritten by r, these symbols specify were to insert the subtrees of the node labeled by s.*

A GEN-grammar is very simple. In fact it describes a morphism of trees. In rewriting of a genetic program, each node is just replaced by a subtree of symbol from S'. Thus the size of the new program (number of nodes) is bounded by a constant K times the size of the old program, where K can be taken as the maximum size of right members of the grammatical rules.

In chapter 4 we present a compiler called JaNNeT that inputs a Pascal program and outputs a neural net that computes what the pascal program specifies. The compiler starts by computing the parse tree of the Pascal program. The parse tree is more complex than an usual parse tree used by a compiler, because it includes nodes for definition of variables. As a result, there is no major differences between this parse tree and a LISP program, modulo a reduction of the LISP instruction set. The second step of the compiler is to rewrite this parse tree into a Cellular Encoding using a GEN-grammar. The third step of the compiler is to develop the Cellular Encoding having chosen an initial value of the life L, this will produce a neural net that computes what the Pascal program specifies, provided that no more than L encapsulated recursive calls are done during the execution. Apart from the development module which is a separate program, the software of the compiler is merely the 27 rewriting rules of the GEN-grammar that rewrites a parse tree into a Cellular Encoding.

We propose a classification of Genetic Language based on this ordering relation.

Definition 6 *Given two genetic language $\mathcal{L}_1$ and $\mathcal{L}_2$, $\mathcal{L}_1$ is higher level than $\mathcal{L}_2$ ($\mathcal{L}_1 \succ \mathcal{L}_2$) if there exists a GEN-grammar able to rewrite any genetic program P written in $\mathcal{L}_1$ into a genetic program P' written in $\mathcal{L}_2$, that does the same computation.*

The existence of the neural compiler JaNNeT proves that Pascal-Parse-Tree $\succ$ Cellular-Encoding. Using a LISP with a reduced instruction set, We have also LISP $\succ$ Cellular Encoding, because our Pascal Parse Trees are similar to LISP S-Expressions, since they include the definition of variables. We now propose other genetic languages, and classify them using $\succ$.

In section 4.4 we show that program-symbols can be micro-coded. The micro coding of program-symbols can be done using a Gen-Grammar. The left member of a rule is the program-symbol name, the right member is the microcode encoded as a tree.

During the software development of JaNNeT, we defined grammar trees of neural building blocks that are called by the compiled code. These grammar trees form a library and can be considered as macros written in Cellular Encoding. Subsection C.7.3 lists the cellular codes of these macros. The genetic language that includes Cellular Encoding plus this library can be called Cellular Macro Encoding. It is possible to derive a Cellular Encoding from a Macro Cellular Encoding using a Gen-Grammar. The left member is the name of the macro, the right member is the code of the macro. Finally, if one uses a Lisp Library, the resulting language can be called a Macro LISP. And is is possible to derive a LISP program from a macro Lisp program, using a Gen grammar. The left symbol is the name of the Lisp function, in the Library. The right member is the LISP code of that function. We can summerize the results with the formula:

Macro LISP $\succ$ LISP $\succ$ Macro CE $\succ$ CE $\succ$ Micro CE.

We now present a concrete example illustrating this hierarchy. Consider the Pascal Program : program p; var a: integer; begin write (a) end. Since variables are set to zero by default, this program will be compiled into a single output neuron with bias 0, that outputs 0. The program has a parse tree which is:

PROGRAM(DECL("a")(TYPE-SIMPLE)(WRITE(IDF-LEC("a"))))

The translation into Macro Cellular Encoding done by the compiler follows. The name of the variables must be replaced with a number encoding the rank of their appearance. Therefore the parse tree should contain variable number instead of variable names. But this is an implementation technical detail of JaNNeT, it does not put into question our hierarchy. It is possible to modify Cellular Encoding in order to keep variable names:

```
IPAR(1)(START-)(SEQ(CPFO(1(SWITCH(1)(IPAR(2)(VAR(LR(1)(VAL(0)(EVER(2))))(PUTL
(2)(BLOC(PAR(CHOPO(1)(SEQ(CPFO(1)(CUTO(1)(CHOP(3)(BPN(1)(END)(MRG(PN-(2))))))
)(BLOC(WRITE-MACRO))))))))(CUT(1)(CUTO(1)))))))))(BLOC(MRG(1)(CUTO(1)))))
```

The translation into Cellular Encoding involves replacing each call to a Macro (here WRITE-MACRO) by the cellular code of the macro, it produces:

```
IPAR(1)(START-)(SEQ(CPFO(1(SWITCH(1)(IPAR(2)(VAR(LR(1)(VAL(0)(EVER(2))))(PUTL
(2)(BLOC(PAR(CHOPO(1)(SEQ(CPFO(1)(CUTO(1)(CHOP(3)(BPN(1)(END)(MRG(PN-(2)))))))
(BLOC((LABEL(-29)(BPN(1)(END)(MRG(1)(SPLIT(JMP(-29)))))))))(CUT(1)(CUTO(1)))))
))))(BLOC(MRG(1)(CUTO(1)))))
```

Because we took a simple program, the tree does not increase a lot between Macro Cellular Encoding and Cellular Encoding. The translation into Micro Cellular Encoding is done by replacing each program-symbol by its microcode, it produces:

```
DIV(m(<)(s(d(=)(m(>)))))(1)(EXE(e)(0))(DIV(s)(0)(-(TOP(m(<)(d(^(#))(m(>=))))(1))
(TOP(m(<)(m($)(m((>$)(m(=))))))(1)(DIV(m(<)(s(d(=)(m(>)))))(2)(AFF(var)(0)(CEL1
(1)(LNK(v)(0)(CEL(e)(2)))))(TOP(m(<)(m(>)(m(=))))(2)(EXE(q)(0)(DIV(d(*)(R(*)))(0
)(-(TOP(m(=))(1))(DIV(s)(0)(-(TOP(m(<)(d(^(#))(m(>=))))))(1)(-(TOP(m(<)(m(>)))(1))
(TOP(m(=))(3)(EXE(p)(1)(EXE(e)(0))(TOP(m(<)(d(^(*))(m(>))))(1)(CEL(e)(2))))))(EXE(
q)(0)(EXE(y)(-29)(EXE(p)(1)(EXE(e)(0))(TOP(m(<)(d(^(*))(m(>))))(1)(HOM(0)(0)(EXE(
j)(-29))))))))(TOP(m(<)(m(>)))(1)(TOP(m(<)(m(>)))(1)))))))))(EXE(q)(0)(TOP(m(<)(
d(^(*))(m(>))))(1)(-(TOP(m(<)(m(>)))(1)))))
```

We can observe that the size the final tree is much bigger, but the necessary number of labels has diminished. It is possible to code the above tree using only 8 different labels.

10.2 Cellular Encoding Versus LISP

Koza has successfully used LISP-like language as a genetic language in a broad range of problems [16]. However, he got impressive results mainly on problems that involve manipulation of symbols. We do not mean mathematical symbols, but entities with a high level meaning. This success may be due to the fact that LISP is a language adapted to symbolic manipulation. On the other hand, when Koza and Rice [15] attempt to search the space of neural nets using LISP, the results are not impressive. They found a neural net for the XOR and for the one-bit adder.

The combination of Genetic Programming with neural network using Cellular Encoding is totally different from the one proposed by Koza and Rice. Here, it is not a program that simulates the neural network, but the neural networks that behave in a program-like manner, since we have shown that a program can be compiled towards a neural net. Modulo this new kind of combination, Genetic Programming of neural networks allows to combine a symbolic search led by the GA, with a connectionist search obtained through learning. We have indeed already combined the genetic search with a learning called Switch learning using a particular kind of combination mode called developmental learning. It is a natural solution to the fundamental problem of building neural networks that can process symbols, without an **EXPLICIT MAPPING** from symbols to neurons or neuron-states. Here, the method exploits all the magic of Genetic Programming that builds a different sort of program without using an **EXPLICIT MAPPING** of a target problem to an algorithm. Another interest of the method is to incorporate modularity into neural networks. Modularity is the property of programming. When considered with neural network encoding, it has been defined in section 3.4. The preceding chapter shows that Cellular Encoding has the property of modularity, and conducts an experiment where modularity is used to solve a difficult problem. Another advantage of Cellular Encoding over LISP is that Cellular Encoding can infer a recurrence where the number of inputs grows. For example, the GA could find a genetic code for a neural network family that computes the parity of n inputs, n arbitrarily large. It seems uneasy to do that with LISP in a natural way, because the inputs must be specified as terminals, which are in finite number.

From the GPII book of John Koza [17] it seems that the most interesting point in GP is to to provide a method for automatic problem decomposition. Like we use a set of grammar-trees, where each grammar-tree encodes a subnetwork, and subnetworks includes themselves in a hierarchical manner, Koza uses a list of LISP S-expression were each S-expression encodes a function and function call each other in a hierarchical way. Koza writes:

> For a variety of different problem, from a variety of different fields, automatic function definition enables genetic programming to automatically and dynamically decompose a problem into subproblems, to automatically and dynamically discover a solution to the sub-problems, to automatically and dynamically discover a way to assemble the solution to the subproblems into a solution to the overall problem.

We write: For a variety of different boolean problems, cellular encoding enables genetic programming to automatically and dynamically decompose a problem into subproblems, to automatically and dynamically discover sub-neural networks that solve the sub-problems, to automatically and dynamically discover a way to assemble the sub-neural networks into a neural networks that solves the overall problem.

In order to use automatic function definition, Koza must define before the run of the GA: the number of functions that will be used, the number of arguments of each of these functions, the allele set for each functions. Cellular encoding does not need any of these. In chapter 9, we used two grammar trees for the two automatically defined subnetworks. In GP, the parameters are explicitly passed to a function so the number of arguments must be known in advance. whereas with CE, the parameters are passed implicitly, by the neurons connecting the cell that will start to develop the sub-neural network to be included. We do not need to define a number of arguments for a subnetwork. We do not need either to define a particular set of alleles for the grammar-trees, although we can do it to help the GA.

The general conclusion of this section is that LISP is a better genetic language to do symbol manipulations, and Cellular Encoding is more adapted for neural network synthesis. We believe that neural network synthesis is a more fundamental problem than symbol manipulation. The root problem of Artificial Intelligence is the symbol grounding problem. It is hard to bridge a symbol to the real world. There is a gap between symbols and the real world which is noisy, fuzzy and very large. If the genetic search is made at the level of symbol, it cannot solve the symbol grounding problem. Genetic search with Cellular Encoding allows to search at lower level of complexities, as we have shown in the previous section, and eventually to define structures like retina or cortical column using macro of Cellular Encoding. Such structures are more in touch with the real world and not directly connected with symbols. A concrete example is the neural network found by Beer and Gallagher that solves a six leg locomotion problem in [2]. They encoded a large network with a few bits, using symmetries. Here the structure is represented by the symmetries, and symmetries can be captured by a Cellular Encoding, using a modular code. If our final aim is to breed complex cognitive structures like a brain, rather than problem solving based on symbol, we believe that it is better to start with a lower level language than LISP.

10.3 Evolving the genetic language itself

Cellular Encoding is a method for encoding families of similarly structured boolean neural networks, that can compute scalable boolean functions. We demonstrate that genetic search of neural networks using Cellular Encoding is equivalent to Genetic Micro Programming and that Cellular Encoding is a micro-programming language of neural-networks. In order to build a theoretical frame that allows generalization, we propose a definition of a Genetic Programming language, together with a criterion to classify their range of complexity. This criterion uses a special kind of very simple tree-grammars called GEN grammars. Five genetic languages have been classified: Macro Lisp $\succ$ LISP $\succ$ Macro Cellular Encoding $\succ$ Cellular Encoding $\succ$ Micro Cellular Encoding. This classification shows that a decomposition is possible from a high level and abstract description based on an efficient programming language down to very simple microcode that specifies simple and local topological modifications of a graph. These modification are duplication or movement of a set of contiguous links. There is no proof as to which of these five languages is the most appropriate for genetic search. We argue that high level genetic languages (LISP) are more adapted to symbolic processing and low level genetic language (Cellular Encoding) to neural network synthesis. In fact the right genetic language itself can be found by genetic search. The initial genetic programs must be built using the most simple genetic language which is Micro Cellular Encoding. A building block is a piece of code that on average increases the fitness of a structure. The symbol of a language at level l is a piece of code at level $l-1$ that has an intrinsic meaning, and therefore, may increase the fitness on average. With this in mind, we can view each symbol of a given language at level l as a building block with respect to the language at level $l-1$. In this scheme, evolution is a process whereby efficient genetic languages of higher complexity are acquired, and stored using a GEN-grammar. The rules of the GEN grammar are the building blocks; they are coded in trees within the chromosome which is an ordered set of trees. The scheme needs an addressing mechanism, a mechanism to decide when to increase the set of trees from n to $n+1$, and a mechanism for initializing the $n+1$ component. The first mechanism can be an addressing by template where the complementary template is searched only on trees of lower order than the calling tree. The second mechanism can increase the number of trees whenever a change in the environment occurs, reflected by a change in the average fitness of the population. The initialization of the $n+1$ component can be simply random. A genetic language found with this method would be a hierarchy of building blocks. If the initial structures are built with a higher level languages than micro-Cellular Encoding, the scheme is a model for the evolution of modular neural networks, where each neural building block of level $l-1$ is a symbol at level l.

10.4 A successful marriage

One constantly oppose artificial neural networks and classical computer based on Turing machine. They are considered as two model of computation, that are very different. A classical computer is programmed with a deterministic algorithm. One knows how much time and memory the computer needs. A neural network is not programmed, it learns by itself. One does not know in advance whether it will succeed to learn, and the number of neurons that are necessary. In this thesis, we marry these two approaches together, using the GA. This marriage is based on the following principles:

Modularity It is not possible to generate automatically big neural net that can solve complex problems, without giving them a modular and hierarchical structure that reflect the regularities of the problem to be solved.

Encoding "Modular" and "hierarchical" can be defined only with respect to an encoding scheme.

Code Optimization The optimization must be done indirectly in the space of codes.

Programming of neural nets An encoding of modular and hierarchical structures is like a programming language for neural network description. A neural network is a labeled graph, from a data structure point of view.

Graph rewriting The encoding must be based on a graph grammar which is the only powerful tool able to describe a graph structure, to our knowledge.

Different possible granularities The encoding may have different possible scaling in its granularities, depending on the underlying graph grammar. Possible granularities are the following:

- One node of the graph is rewritten in many nodes
- One node of the graph is rewritten in 0,1 or 2 nodes
- list of links are duplicated or moved

Specification of the learning Classical learning in neural networks operates on a fixed size architecture and is based on a gradient descent. Classical learning must converge to a global optimum. Now, it must be considered as a tool for accelerating the research which is done in the space of code.

In this thesis, these new paradigms where illustrated as follows:

Modularity Experiments shows that it is possible to automatically generate huge neural networks, that are modular, for solving arbitrary large boolean problems.

Encoding A formal and precise definition of modularity has been given. It was shown that cellular encoding is modular.

Code Optimization Experimental results show that the Genetic Algorithm is an efficient optimization procedure. Moreover, the building block hypothesis that ensures the success of the GA can be interpreted: A sub neural network is encoded in a sub tree of the cellular code. The sub tree will be a building block, if the sub neural network has some interesting features. When this sub neural net is included into a bigger neural net, the fitness of the whole neural net will be good on average.

Programming of neural networks Properties of the coding as well as the existence of the neural compiler JaNNeT show that the cellular encoding is a kind of programming language for neural network description.

Graph rewriting Cellular encoding encodes a parallel graph rewriting system.

Different granularities The macro program symbols that have been introduced for the neural compiler, the program symbols of cellular encoding and the micro coding, illustrate the different possible granularities.

Learning We have defined a new learning, called switch learning, as well as a new way to combine learning and the research led by the Genetic Algorithm called developmental. Switch learning combined with the developmental mode allows to achieve a speed up of 13.

The marriage between programming and neural network, present both schemes under a new light. For examples: we are no longer obliged to use neurons with a traditional sigmoid, we can use neurons that make the product of their inputs. The genetic algorithm can generate any type of neurons. Instead of learning all the weights, one can design neural net where only part of the neural net is learned, and the other part is compiled, or directly generated with cellular encoding. The specification of the learning has changed, instead of to converge to the global optimum, the aim of the learning is now to accelerate the genetic search. It is not so much important to parallelize learning of the neural net. The parallelism is more natural, and more powerful at the level of the genetic search. The genetic algorithm has a super linear speed up. It is the fitness of a code that costs a lot of time to be evaluated. and the fitnesses can be evaluated in parallel.

The idea to optimize neural networks indirectly, using a code, is a deep change of paradigm. We are certain that sooner or later, this new principle will change the way people are now looking at neural networks. The real problem in artificial intelligence, or in the connectionism, is a scaling problem. Methods are successful on a small scale, but they have an exponential complexity, and take too much computations, when the size of the problem become big. The only way to solve the scaling problem, is to provide an algorithm that can automatically decompose a problem into subproblems, solve each subproblem, and assemble the solutions of the sub problems into a general solution of the whole problem. The solution of a problem solved with such an algorithm had automatically a modular and hierarchical structure that reflects this decomposition. John Koza [17] demonstrate on a number of examples, that genetic programming can achieve such an automatic decomposition and recomposition. Koza uses what he calls Automatic Function Definition. (ADF) The GA automatically decomposes a problem into sub problems, generate functions (ADF) that solve these sub problems, and use many times function calls to recompose a solution to the whole problem. The book GPII from John Koza is a real breakthrough. Nevertheless, Koza did not think that his concept of automatic decomposition and recomposition could be applied to neural nets. Koza writes:

> "Conceivably the subtask performed by the neuron might be useful elsewhere in the neural network. However, existing paradigms for neural networks do not provide a way to re-use the set of weights discovered in one part of the network in other part of the network where a similar subtask must be performed on a different set of inputs"

It is precisely what the GA combined with cellular encoding can do. It can reuse not only a single neuron, but a sub network, likewise genetic programming can reuse a procedure. For a variety of different problems, cellular encoding allowed the GA to automatically and dynamically decompose a problem into sub problems, build a sub network for solving the sub problem, and discover a way to assemble multiple copies of that subnetwork in order to build a network for the global problem. In the three experimental results: parity symmetry, and decoder, the neural net structure exactly reflects the problem structure.

A huge work is still to be done, in order to apply the method proposed in this thesis to other problems than boolean functions. Koza has shown that automatic function definition becomes interesting when the problem to be solved has an amount of regularities above a break even point. Any neural network application where the neural net have a certain amount of regularities can be an interesting application for cellular encoding. We have two particular propositions: In [2], Beer and Gallagher have generated with the GA, a neural network for the locomotion of a 6 leg robot. They used the symmetries of the problem in order to obtain a compact representation. The neural net is composed of six copies of the same sub neural network: $N_i, i = 1, .., 6$, each one corresponds to one leg. Connections between these sub networks are shaped like an 8, They are also identical.

Beer and Gallagher have encoded on a chromosome a single sub network N_1, and the connections between N_1 and N_2. Thanks to this highly compact representation, the problem could be solved. Cellular encoding can automatically exploit the symmetries of the problem to provide a compact encoding. Part of the code represents N_1, and another part specifies how to assemble 6 copies of N_1 in order to build the whole network. With cellular encoding, the GA can automatically decompose the problem into 6 identical sub problems, generate a solution adapted to each of this 6 sub problem and assemble 6 copies of this solution. The problem for the locomotion of the 6 legs robot can be solved automatically, without using the knowledge we have about the symmetries of the problem. The other application concerns neural nets that are translation invariant, used for low level visual processing. Here, the same sub network is repeated as many times as there are pixels. It represents a building block encoded separately, like a procedure.

This thesis presents novel ideas that need many parameters in order to be implemented in a computer. I admit that sometimes, I do not remember how to set all these parameters, which can be more than one hundred. The usual question in computer science: "How shall we do to go faster" must be studied in a systematic way. Going faster is the constant motivation of all the simulation that we have done. We always wanted to improve the performance of the genetic algorithm in order to solve problems of increasing complexities. We looked for an optimal learning with this goal in mind. In the first chapter of the second part, chapter 6, the GA takes around 800 seconds to solve the parity problem. This time was mentioned in the first publication on cellular encoding, in COGANN92. This time is reduced to 1.2 seconds in chapter 9, using switch learning, combined with a developmental mode, and the mixed parallel GA on a 64 nodes IPSC860 computer. The speed up is 660. We are convinced that with further research, and using modern parallel computers it is possible to go 1000 times faster than we are now doing. This race is not a game but a need. During one hour, one must generate, train and test over one million of neural nets, having tens of neurons, as in the decoder experiments. Going faster is the only way to obtain a robust method for automatic generation of huge and modular neural networks.

Bibliography

[1] D.H. Ackley and M. Littman. Interactions between learning and evolution. In *the 2nd Conf. on Artificial Life*. Addison-Wesley, 1991.

[2] Randall Beer and John Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1:92–122, 1993.

[3] R.K. Belew. When both individuals and populations search: Adding simple learning to the genetic algorithm. In D. Schaffer, editor, *3th Intern. Conf. on Genetic Algorithms*. Morgan Kaufmann, 1989.

[4] R. Collins and D. Jefferson. Selection in massively parallel genetic algorithm. In *4th Intern. Conf. on Genetic Algorithms*, 1991.

[5] G.E. Hinton D.H. Ackley and T.J. Sejnowski. A learning algorithm for boltzman machines. *Cognitive Science*, 9:147–169, 1985.

[6] H. Ehrig, M. Korf, and M. Lowe. A tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In *LNCS 532*, 1990.

[7] Max Garzon and Stan Franklin. Neural computability ii (extended abstract). In *IJCNN*, 1989.

[8] M. B. Gordon, P. Peretto, and D. Berchier. Learning algorithms for perceptrons from statistical physics. *Journal de Physique*, january 1993.

[9] S. Harp, T. Samad, and A. Guha. Toward the genetic synthesis of neural networks. In D. J. Schaffer, editor, *3rd Intern. Conf. on Genetic Algorithms*, pages 360–369, 1989.

[10] G.E. Hinton and S.J. Nowlan. How learning can guide evolution. *Complex Systems*, 4:495–502, 1987.

[11] John Holland. *Adaptation in natural and artificial systems*. MIT press, 1993.

[12] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

[13] J.M.Baldwin. A new factor in evolution. *American naturalist*, 30:441–451, 1896.

[14] H. Kitano. Designing neural network using genetic algorithm with graph generation system. *Complex Systems*, 4:461–476, 1990.

[15] J. R. Koza and J. P. Rice. Genetic generation of both the weigths and architecture for a neural network. In *IJCNN92, International Joint Conference on Neural Networks*, pages 397–404, 1991.

[16] John R. Koza. *Genetic programming:On the programming of computers by mean of natural selection.* MIT press, 1992.

[17] John R. Koza. *Genetic programming II: Automatic Discovery of Reusable Subprograms.* MIT press, 1994. to appear.

[18] Tsu-Chang Lee. *Structure level adaptation for artificial neural networks.* Kluwer Academics Pubishers, 1991.

[19] Alexander Linden and Christoph Tietz. Combining multiple neural network paradigms and applications using sesame. In *International Joint Conference onNeural Netwroks, Baltimore*, 1992.

[20] A. Lindenmmayer. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, 18:280–299, 1968.

[21] S. Lucas. An algebraic approach to learning in syntactic neural networks. In *IJCNN92*, 1992.

[22] Eric Mjolness, David Sharp, and Bradley Alpert. Scaling, machine learning and genetic neural nets. La-ur-88-142, Los Alamos National Laboratory, 1988.

[23] H. Muehlenbein. Limitations of multi-layer perceptrons networks - steps towards genetic neural networks. *Parallel Computing*, 14:249–260, 1990.

[24] H. Muehlenbein. Darwin's continent cycle theory and its simulation by the prisonners dilemma. *Complex System*, pages 459–478, 1991.

[25] H. Muehlenbein, M. Schomish, and J.Born. The parallel genetic algorithm as function optimizer. *Parallel Computing*, 17:619–632, 1991.

[26] Hans P.Zima, Peter Brezany, Barbara Chapman, and Jan Hulman. Automatic parallelization for distributed-memory systems: Experiences and current research. In *European Informatics Congress Systems Architecture*, 1993.

[27] Thomas Ray. Is it alive or is it ga ? In Richard Belew, editor, *4th Intern. Conf. on Genetic Algorithms*, pages 527–534, 1991.

[28] J.D. Schaffer and D. Whitley . *Combination of Genetic Algorithms and Neural Networks.* IEEE Computer Society press, 1992.

[29] Hava Siegelman and Eduardo D. Sontag. Neural nets are universal computing devices. Sycon-91-08, Rutger University, 1991.

[30] R. Tanese. Distributed genetic algorithm. In D. J. Schaffer, editor, *3rd Intern. Conf. on Genetic Algorithms*, pages 434 – 440, 1989.

[31] D. Whitley. The genitor algorithm and selection pressure why rank based allocation of reproductive trials is best. In D. J. Schaffer, editor, *3rd Intern. Conf. on Genetic Algorithms*, pages 116–121, 1989.

[32] D. Whitley, T. Stakweather, and C. Bogart. Genetic algorithms and neural networks, optimizing connection and connectivity. *Parallel Computing*, 14:347–361, 1990.

Appendix A

Personnal Bibliography

A.1 Reviewed articles

- Adding learning to the the cellular developmental process: a comparative study. *Evolutionnary Computation*, 1993. V1N3 pp213 233. In collaboration with Darell Whitley
- Genetic Microprograming of Neural Networks Contribution au livre *Advance in Genetic Programing* edited by K. Kinnear, MIT press 1993, to appear.
- Genetic programing of Neural Networks, theory and practice Contribution au livre *Intelligent hybrid systems* edited by S. Goonatilake and S. Khebbal, John Whiley, 1993 to appear.

A.2 Patent

Patent entitled: Demande de Brevet Français EN 93 158 92 December 30, 1993, Process of translation and conception of neural networks, based on a logical description of the target problem.

A.3 Conference with published proceedings

- A 2D toroidal systolic array for the knapsack problem, In *Algorithms and Parallel VLSI Architectures II* 1991. Elsevier In collaboration with Rumen Andonov
- A modular systolic 2D torus for the general knapsack problem. In *Application Specific Array Processors* 1991 IEE Computer Society Press. In collaboration with Rumen Andonov
- Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In *Combination of Genetic Algorithms and Neural Networks*, 1992. IEEE Computer Society press
- Genetic synthesis of modular neural networks. In *5th International conference on Genetic ALgorithm*, 1993. Morgan Kaufman Publisher
- Grammatical inference with genetic search using cellular encoding. In *Grammatical Inference: Theory,applications and alternatives*, 1993. Digest No 1993/092
- A learning and pruning algorithm for genetic neural networks. In *European Symposium on Artificial Neural Network*, 1993.

- The mixed parallel genetic algorithm. In *PARCO93 (Parallel Computing)*, 1993.

A.4 National Conference with published proceedings

Un outil pour combiner approche symboliques et connexionistes. In *Emergence des symboles dans les modéles de la cognition*, 1993. Editor: Bernard Amy, Alain Grumbach Jean-Jacques Ducret, in collaboration with Jean-Yves Ratajszcak and Gilles Wiber.

A.5 Research report

- Cellular encoding of genetic neural network. research report 92.21, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1992.
- Adding learning to the the cellular developmental process: a comparative study. Research report rr93-04, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1993. In collaboration with D. Whitley.
- Compilateur Pascal de Réseaux de Neurones Rapport de stage de projet de fin d'année Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquée de Grenoble in collaboration with Jean-Yves Ratajszcak and Gilles Wiber.

A.6 Conferences without proceedings

- A neural Compiler using Graph Grammars. oral presentation Journees ASMICS , Schloss Dagstuhl, Octobre 1993. (Algebraic and Syntactical Methods in Computer Science)
- Genetic algorithm for designing neural network. Journees Modélisation et Cognition, Autrans, Pole Rhone Alpes de Cogniscience, 1992.
- Equivalence between cellular encoding and graph grammar. oral presentation Journees ASMICS , Bordeaux, 1993. (Algebraic and Syntactical Methods in Computer Science)
- Parallel GA applied to neural network synthesis. Journees Modélisation et Cognition, Paris, 1993.
- A parallel genetic algorithm. oral presentation at Journées CAPA, Grenoble, 1993. (Conception et Analyse des Algorithmes Parallèles)
- Participation to the summer school "Mathematical methods for artificial neural networks" at the Kurt Bösch Institute in Sion (Switzerland) 1992.

Appendix B

Cellular Encoding is a graph grammar

B.1 Introduction

In this chapter, Cellular encoding is discussed in relation to graph grammars. Cellular Encoding is specified as a system of rewriting rules. The objects that are rewritten are *cells*, where a cell is a node of a directed graph, with **ordered** connections. If the rewriting rules are interpreted recursively i times, they develop the i_{th} neural-net of the family, that should compute the parity of i inputs, if parity is the target problem. In order to unify cellular encoding with the concept of usual graph grammar, it is shown that a cellular encoding can be translated into a graph grammar of the kind used in the Berlin algebraic approach [6]. The process of encoding a grammar into a set of trees is described. Trees are an interesting data structure, because the genetic algorithm can be efficiently used to recombine and mutate them. Experimental results obtained on the symmetry, the parity and the decoder boolean function, in the second part of this thesis, support this claim.

Searching a cellular encoding with the GA is a grammatical inference process. It differs from usual grammatical inference approach because the searched grammar does not have to generate a language specified by positive and negative examples. Here the language is specified indirectly. The language of neural networks must be such that the i^{th} neural net computes the mapping for the target problem of size i, and must be of reasonable size.

Our approach is different from the "syntactic neural network" of Lucas [21], where the language is itself the learning task, and therefore, it is known from positive and negative examples. In Lucas's approach, there seems to be an "isomorphism" between the grammar and the syntactic neural network, i.e, a grammar is mapped on a neural network. Syntactic neural networks have always the same type of architectures, whereas cellular encoding does not impose a prior constraint on the architecture, neither structural constraint nor complexity constraint.

B.2 Cellular Encoding presented as a graph grammar

Cellular encoding can be seen as a system of rewriting rules. These rules operate on *cells*. A cell is a node of a directed *network graph* with ordered connections. Each cell is labeled with a symbol of the grammar alphabet, either terminal or non-terminal. There is a single terminal celll which is labeled by n. Terminal cells are finished neurons.

Cells and connections have attributes, some of which are the thresholds and weights of the final neural net, while others are used by the rewriting rules. For example, the *link-register* is used to refer to one of possibly several fan-in connections into a cell. The *life* attribute is used to bound the number of recursive interpretations of the grammar.

A rule r is denoted $A \overset{F}{\rightarrow} \alpha$ rewrites one cell into k cells, where k is the lenght of the string α. A is a non terminal symbol that must label the *parent cell* on which r is applied, α is a string of k symbols, terminal or non terminal that label the k *child cells* created by the rule, and F refers to a program-symbol for computing the links and attributes for each of the k child cells. Here we consider only the case $k \leq 2$. If $k = 2$, the number of cells of the network-graph augments by one. The parent cell is said to *divide* into two child cells. In this case, the attributes of the two child cells are inherited from the parent cell. If $k = 1$, the rewriting rule *modifies* the attributes of the parent cell. If $k = 0$, the cell is deleted.

The axiom of the grammar is a network graph that consists of a single cell called the *ancestor cell* connected to an input *pointer cell* and an output *pointer cell*. Consider the starting network and the encoding depicted in figure B.1. At the starting step, the attributes of the ancestor cell are initialized with default values. The threshold is 0, the link-register is 1, the life is L, if we want to develop the L^{th} network. As this cell repeatedly divides, it gives birth to all the other cells that will eventually make up the neural network. The input pointer cell and the output pointer cell are not rewritten. They are used to point to the input and output unit of the network. For example, in figure B.1 (a), the XOR neural net has two input units labeled 1 and 2, and one output unit labeled 1. There are many kinds of cell division and cell modification, depending on the particular program-symbol used to compute the attributes and links of the child cells. We illustrate an example in figure B.1. The grammar used is $\{\$ \overset{SEQ}{\rightarrow} AB,\ A \overset{PAR}{\rightarrow} \$n,\ B \overset{SEQ}{\rightarrow} CD,\ C \overset{PAR}{\rightarrow} En,\ E \overset{INCBIAS}{\rightarrow} n,\ D \overset{VAL-}{\rightarrow} n\}$. The program-symbol `SEQ, PAR, INCBIAS, VAL-` have been defined in the preceding chapter.

The sequence in which cells are rewritten is determined as follows: once a cell is rewritten, it enters a First In First Out (FIFO) queue. The next cell to be rewritten is the head of the FIFO queue. If the cell divides, the child which reads the first letter of the right member enters the FIFO queue first. This order of execution tries to model what would happen if cells were rewritten in parallel. It ensures the following invariant:

> **Invariant 1:** A cell cannot be rewritten twice while another cell has not been rewritten at all.

By imposing a rewriting order we ensure that a grammar produces a single network for a given initial life value L. The handling of L is done as follows: each cell manages a stack of characters, used to record the history of its rewriting. Each time a rewriting rule like $\$ \overset{SEQ}{\rightarrow} AB$ is applied on a cell, the cell executes a 3-steps algorithm. Call s the symbol in the left member, here $s = \$$. First the cell checks whether s appears on its stack. If it does, the cell decrements its life number and empties its stack. Second, the cell pushes s on its stack. Third, the cell checks whether its life is strictly positive. If $L > 0$ the rule $\$ \overset{SEQ}{\rightarrow} AB$ is used, else the rule $\$ \overset{WAIT}{\rightarrow} n$ is used, and the cell becomes a finished neuron.

B.3 Encoding a Grammar into a set of trees

In this section is described how to encode a “clean” grammar into a set of trees. A clean grammar means a deterministic grammar where all the symbols are reached by the productions. First,

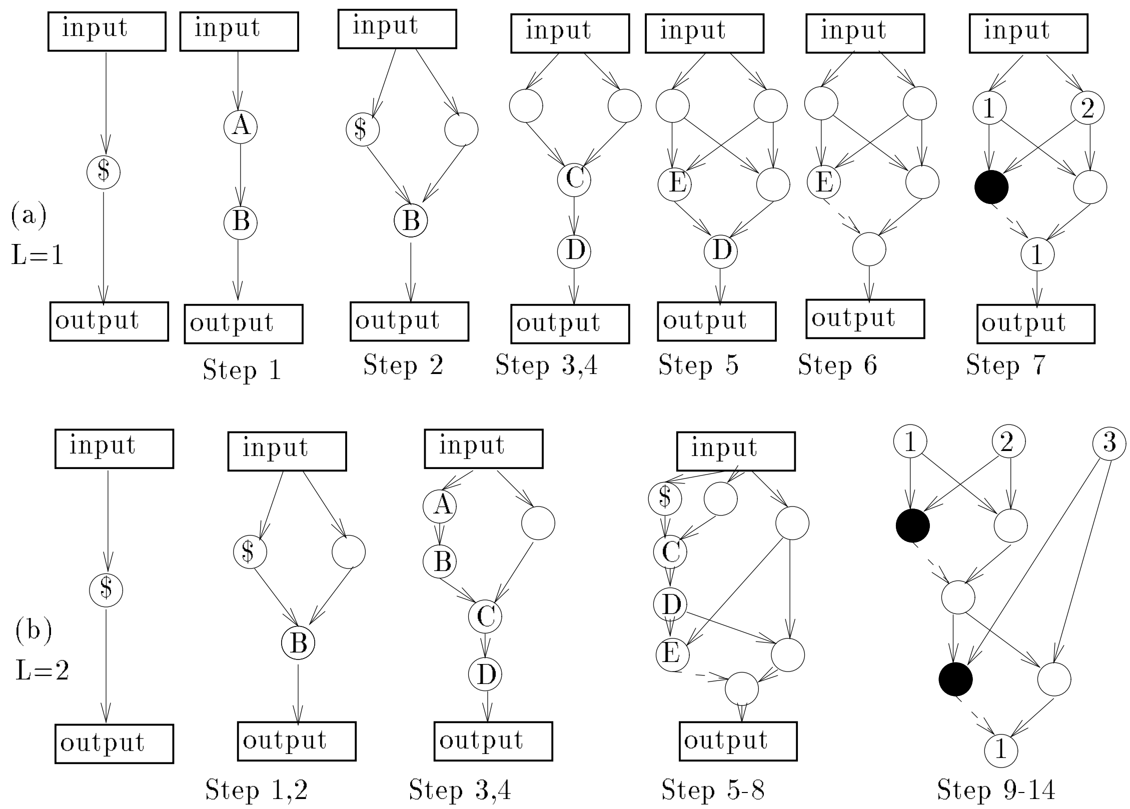

Figure B.1: The derivation of a parity network. The box labeled input represents the input pointer cell, The box labeled output represents the output pointer cell; Circles represent other cells. If the threshold is 0, the circle is empty, while if the threshold is 1 the circle is filled with black. A continuous line indicates a weight 1, and a dashed line a weight -1. The first seven network graphs (a) show the derivation for L=1, the network computes the XOR. In the second sequence of derivation (b), starting with L=2, the network for the three inputs parity is derived.

a directed graph $\mathcal{G}$ that represents the grammar is built in a constructive manner. Initially $\mathcal{G}$ contains one node for each non-terminal, labeled by the name of the corresponding non-terminal. For each rule $A \xrightarrow{F} \alpha$ where α includes the non terminal B, an edge is drawn from the node A to the node B. If α includes the terminal n, a new node labeled n is created and an edge is drawn from the node A to this node. The final graph has therefore as many nodes labeled with n as there are occurrences of n in right members of the rules. Since the grammar is clean, it is possible to embed a tree $\mathcal{T}$ in $\mathcal{G}$ such that the root of the tree is labeled by the axiom, and $\mathcal{T}$ contains all the nodes of $\mathcal{G}$. We now want to decompose the graph $\mathcal{G}$ into a set of trees. Branches of $\mathcal{T}$ are pruned. They become a tree once they are pruned, and are given a name which is a lower case. The initial tree $\mathcal{T}$ is a particular branch named with the symbol \$. The pruning of branches is done by processing $\mathcal{T}$ in a bread first manner. Each time a node N of $\mathcal{T}$ has some k input links in $\mathcal{G} \setminus \mathcal{T}$, these k links are redirected towards k newly created node of arity 0. There are two possible cases for the labeling of these created nodes. If N is the root of $\mathcal{T}$, the k nodes are labeled with the name of $\mathcal{T}$. Else, the branch of $\mathcal{T}$ beginning at N is pruned, and given a name x, and the k added nodes are labeled by x. In the last case, node N in $\mathcal{T}$ is replaced by a 0-arity node also labeled by x. Once $\mathcal{T}$ is fully processed, the pruning procedure is recursively applied to the pruned branches. The recursive provess will end, because each time it is applied, the number of nodes of the branches to be processed, strictly diminishes. When the process is ended we obtain a set of branches or trees. In this resulting set of trees, each non zero-arity node is labeled with a non terminal. We replace this non terminal by the name of the program-symbol of the rule that rewrites it. In this way, the final set of trees can be interpreted without any references to the initial grammar.

Each lower case x is to be interpreted as a `JMP x` program symbol, that we have seen in chapter 4. A cell that executes the program-symbol x just places its reading-head on the root of the grammar-tree which name is x. The program-symbol `REC` used in section 2.2 is equivalent to a program-symbol x where x is the name of the tree that is curently read by the cell. The register life of the cell L, must be decremented, not only in case of a `REC` program-symbol, but also to prevent an infinte looping between two sub-tree that call each other. This is the reason why each cell manages a stack of characters, used to record the history of its rewriting. The letter "n" must be interpreted as an `END` program symbol.

Figure B.2 shows how to transform the grammar for the parity network into a set of 1 tree. The grammar $\{\$ \stackrel{SEQ}{\rightarrow} AB, A \stackrel{PAR}{\rightarrow} An, B \stackrel{PAR}{\rightarrow} BB\}$ is also translated into a set of 3 trees. This last grammar derives a 2 layers network with L input units and 2^L output units. Each input unit is connected to all the output units. Since there is no need to code names of non-terminal, the encoding of a grammar as a set of trees is more compact if there are few trees. In the case of the parity grammar, the encoding as a set of rules needs $\lceil \ln(6^{4*3+2*2} * 4^6) \rceil = 55$ bits, whereas the encoding as a tree need only $\lceil \ln(6^{11}) \rceil = 29$ bits, that is about the half. Encoding a grammar in a single tree ensures that the grammar that are generated are always clean. It is a good closure property for the application of the GA. More and more people in the GA-community tends to put information into the structures manipulated by the GA, to ensure that these structures are similarly kept "clean".

B.4 Equivalence with the algebraic approach to graph grammar

In this section, we show that a cellular encoding grammar can be translated into a graph grammar of the kind proposed by Ehrig, Körff and Löwe so-called "Berlin algebraic approach". In [6] they write:

> The rewriting of a graph G, via some rule p is done by first deleting some part DEL and then adding a new part ADD finally resulting into a new derived graph H. A rule p shall only be applicable if G additionally contains some context $K_L$which is essentially to be kept. Rewriting means to replace some part $L = DEL \cup K_L$ by another one $R = K_R \cup ADD$

A simplified model is enough for our purpose. The graph DEL will always be a single node n, $K_R = K_L$ will be all the neighbor nodes of n. Therefore, the representation of a production by the pair (L, R) will be easy to understand. Moreover, ADD will most of the time include only one or two nodes.

In the network-graph used in cellular encoding the links are ordered. A network graph can be represented as an ordinary directed-graph where the links are not ordered, as indicated in figure B.3. Each cell corresponds to a node represented by a circle, each link l connecting a cell c_i to a cell c_o, corresponds to two squares s_i and s_o and one oval. Cell c_i (resp. c_o) uses the square s_i (resp. s_o) to encode the link l considered as an output link of c_i (resp. an input link of c_o). The oval

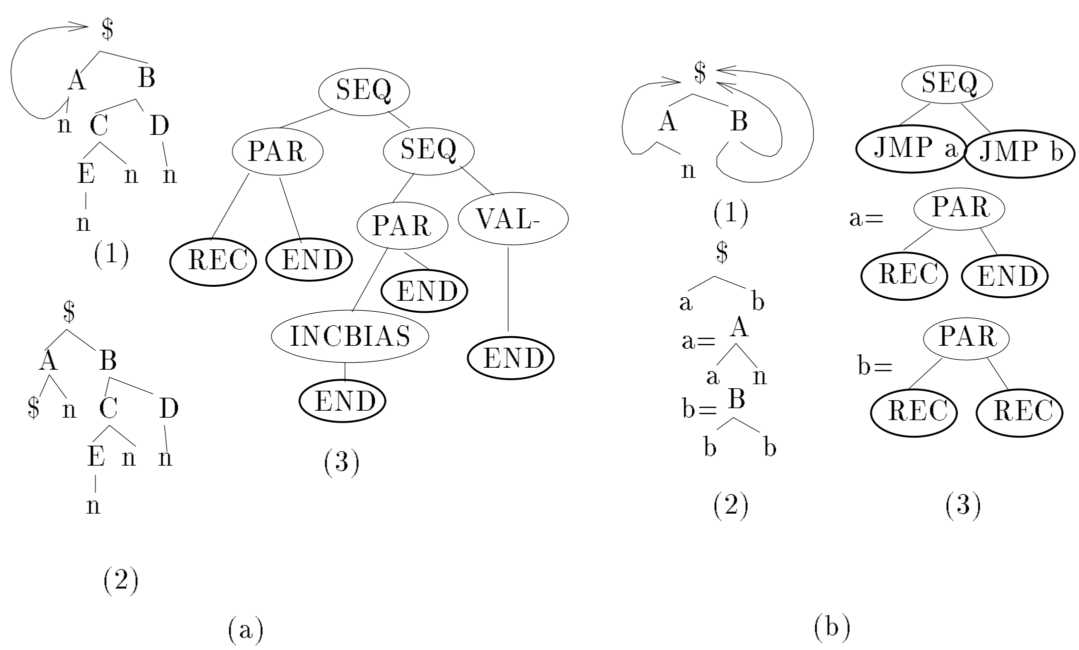

Figure B.2: Transforming a grammar into a set of trees. (a) A grammar for the parity (b) A grammar for a 2 layers network with L input units and 2^L output units. (1) Shows the graph $\mathcal{G}$ and the embeded tree $\mathcal{T}$. (2) Shows the results of the pruning. (3) Presents the final set of trees, labeled with the names of the program-symbols.

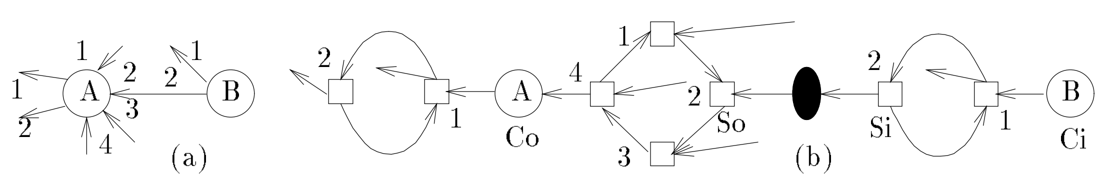

Figure B.3: How to represent a network-graph as a directed graph. The link-register of the cell labeled A points to the 4^{th} input link.

connects s_i to s_o. The squares representing the list of input-links, (or output links) of a given cell are organized in a 1-D torus. The circle representing the cell is connected to a particular square of the input list. This square corresponds to the link pointed by the link-register. The fan-in and fan-out of this directed graph is bounded by 2. The edges that fan-in or fan-out from a node need not be ordered, as was the case with the network-graph representation. The circles are labeled with non-terminal of the grammar. Ovals and squares need not be labeled. The edges may be labeled with arrows that indicate the next node to be rewritten, as will be later explained. Each production of the grammar will be translated into a corresponding production on the associated directed graph, called G.

The sequential division in figure B.4(a) can be easily translated, because its effect remains local on G. However, the parallel division is not obvious. We write the rule assuming that the cells have more than three inputs and outputs. The case where the linked list of squares has one or two squares are special cases. We do not write the rules for these special cases because they are similar to the rules in figure B.4.

The first rule of (b) duplicates the circle-node representing the cell. The next three rules 2-4 of figure B.4(b) are used to duplicate the squares corresponding to the the links of the parent cell, The 5^{th} rule describes how to duplicate the ovals. The following two rules 6-7 show how to duplicate the squares corresponding to the 1-D torus of squares of the neighbors. The neighbor cell might also have done a parallel division in this case the oval will be replicated in four, as specified by the 8^{th} rule. The oval must not be rewritten before knowing whether the cells linked through this oval duplicate or not. Edges are labeled with arrows used for this purpose. The arrows circulate through the 1-D torus of squares, two round tours for the simulation of one rewriting step. Rule 9,10,11 account for the beginning, the middle, the end of the first tour. After the first tour, the edges connecting the ovals are labeled with arrows oriented towards the ovals. Rules 9,10,11 need not be used in the case of parallel division, where the arrow on edge connecting the oval are set during the square duplications. Rules 9,10,11 are used for the sequential division (with rules 0). As shown by rule 5 or rule 8, an oval can be rewritten only when all its connections are labeled with incoming arrows. This ensures that the effect of all neighbor division has been taken into account. Rules 11,12,13 specify the beginning, middle and the end of the second tour. After the second circulation of arrows, the 1-D ring of squares is updated, and ready to start simulation of a new rewriting. The cell can apply the next rewriting rule only when both the list of squares that fans-in and fans-out have been updated. When it is the case, there are two arrows oriented towards the circle representing the cell. The circulation of arrows has another interest. It ensures that the rewriting order will be the same in the original grammar and in the graph grammar by allowing to check Invariant 2. This invariant is not exactly the same as invariant 1, but it ensures at least that both grammars derive the same network.

> **Invariant 2**: A cell cannot be rewritten twice while one of its neighbor cell has not been rewritten at all.

We now translate the rules that modify cell attributes. It is not difficult to increment or decrement the link-register. We just need to rotate the ring of squares, one square forward or one square backward. Rule 1 of figure B.5 describes how to increment the link register. Rule 2 specifies how to set the weight of the link pointed by the link register to -1. Rule 3 shows how to cut the link pointed by the link register. Rule 4,5 make the oval disappear. Rule 6,7 simplify the neighbor's linked list of squares.

Rules 4,5,6,7 figure B.5 (b) handle the flow of arrows. They do not perfectly model the previous order of execution used to develop a cellular encoding. In order to match the new rules, this order of execution must be changed for the sequential division. The new definition must be the following: If the cell divides in a sequential way, the child which reads the first letter of the right member enters the FIFO queue last (instead of first). It comes from the fact that a cell can modify its input links, but not its output links. Consider two cells c_1 and c_2 linked through a link l from c_1 to c_2. Assume c_1 makes a parallel division which makes the link l duplicate into two links numbered l and $l+1$. Let c_2 cut (or modify the weight of) l. Then translate the cellular encoding into a graph grammar. How will the network-graph be rewritten by the graph grammar? The links l and $l+1$ of c_2 will be cut (or have their weight modified). With the modification brought to sequential division, the order of rewriting of the cell will be the closure of an order induced by the directed acyclic graph of cells, where a cell is rewriten after another one if it is linked to it.

Until now we have translated everything except the managing of the life attribute. It is possible fit a more classical frame of rewriting system, by copying the approach proposed by Lindenmeyer's L-system [20]. In L-systems, the rewriting rules are applied in parallel. Let us also apply the rewriting rule of a grammar resulting from the translation of a cellular encoding, in parallel. Write the cellular encoding as an ordered set of trees. Assume that a tree numbered i can only refer to other trees numbered j, where $i \leq j$. Add a number of waiting operator so that the same number of rewriting steps C is needed to go through the different trees. The derivation for a given L value can be reproduced by the parallel application of the rules. The rewriting must be stopped at time $C * (L + i)$ for the tree i, and it is no more needed to handle a life attribute.

B.5 Conclusion

Cellular Encoding is a method for encoding neural network families into a set of named trees. This chapter shows that cellular encoding encodes a graph grammar. More precisely, we make precise how to translate a cellular encoding into a set of graph grammar rewriting rules of the kind used in the "Berlin algebraic approach to graph rewriting". This translation is done for a restricted alphabet of the cellular encoding code, and still need to be done for the whole alphabet listed in appendix C.5. We believe that this should not involve conceptual difficulties. The equivalence between cellular encoding and graph grammar brings a new light on genetic neural network. The genetic search of neural networks via cellular encoding appears as a grammatical inference process where the language is implicitly specified, not explicitly by positive and negative examples. This results in three new perspectives: We will use a method to search cellular encoding (second part of this thesis) that is, to encode a grammar into a set of trees and evolve sets of trees with the genetic algorithm. This chapter shows that this method can be used to infer any kind of grammars. Conversely, result from the research in the grammatical inference field can be used to improve cellular encoding. Last, the study of cellular encoding as a grammar can be helpful to classify cellular encoding into the complexity classes that already exist among rewriting grammars. It seems that cellular encoding is a context sensitive, parallel graph grammar.

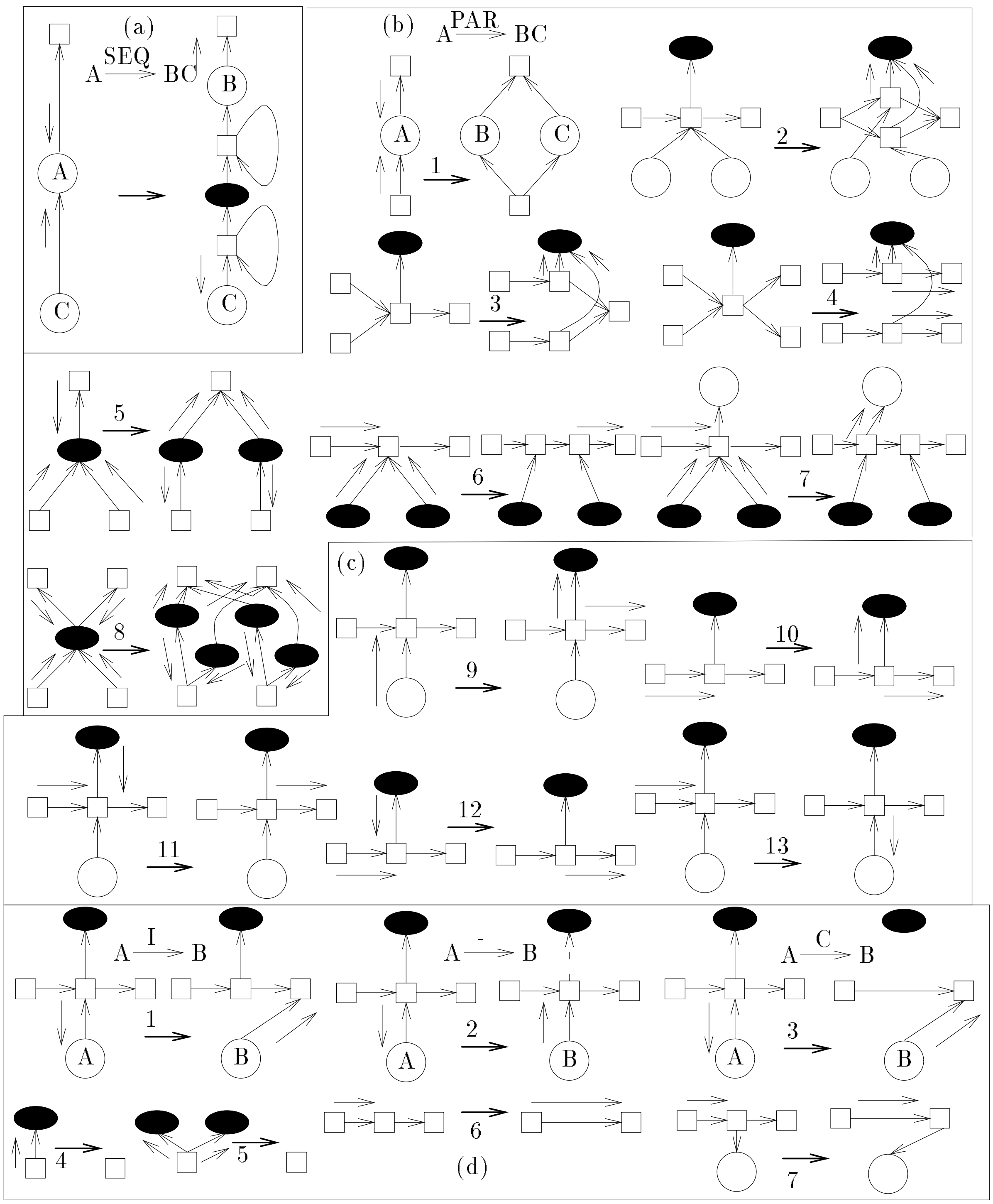

Figure B.4: Translation of the rule for cell division. (a) The sequential division (b) The parallel division (c) Managing of the edge labels (arrows).

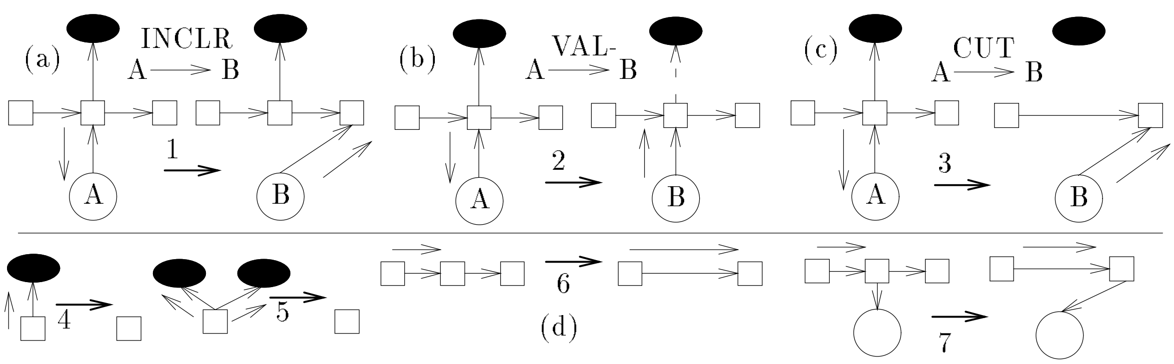

Figure B.5: Translation of the rules for cell modification

Appendix C

Technical Complements

C.1 Small example of compilation

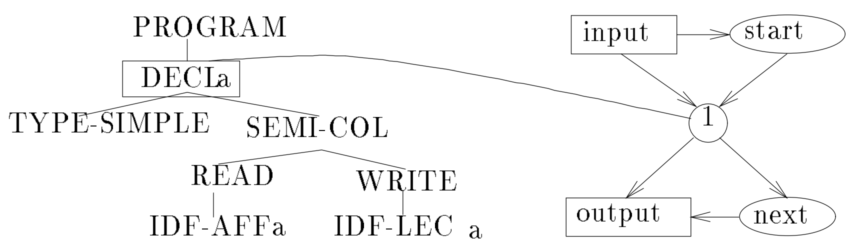

Figure C.1: Execution at step one of the macro program symbol `PROGRAM`. The ancestor cell gives birth to two other cells. A cell labeled "start" and another one labeled "next". The "start" cell is used to start the neural net, and to manage the propagation of the activities. The "next" cell has not finished its development. It waits that all its neighbors become finished neurons. (i.e. that they have lost their reading head.) The ancestor cell now reads the `DECL` node.

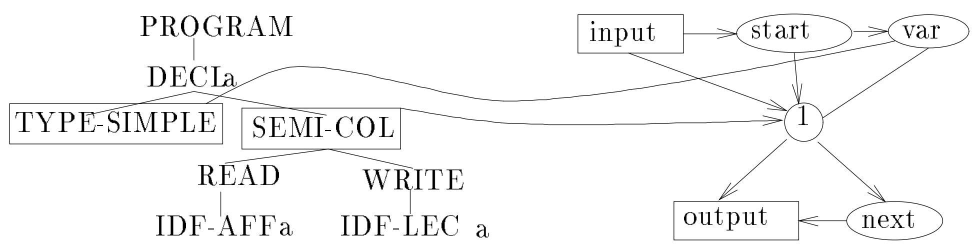

Figure C.2: Execution at step 2 of the macro program symbol `DECL`. The ancestor cell gives birth to another cell labeled "var". This cell represents the variable "a". The input and output connections of the ancestor cell are now complete and represent a topological invariant. This invariant will be checked by a cell, whenever a cell executes a macro program symbol. The first input link points to the input pointer cell. The second input link points the start cell, the rest of the input links point to neurons that contains the value of the variables in the environment. The first output link points the output pointer cell. The second output link points the "next" cell which is used to establish connections to the rest of the neural net.

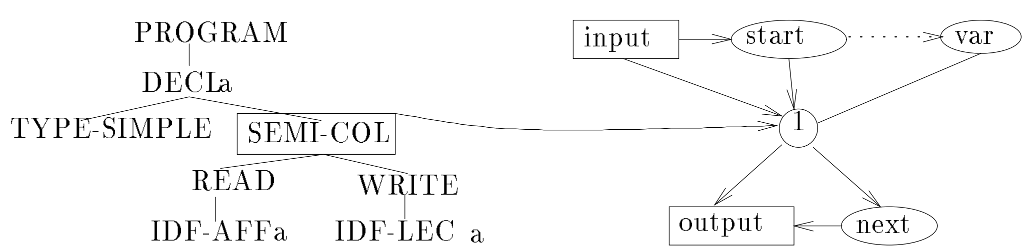

Figure C.3: Execution at step 3 of the macro program symbol `TYPE-SIMPLE`. The effect is to set to the value 0, the weight of the connection to the neuron that represents variable "a".

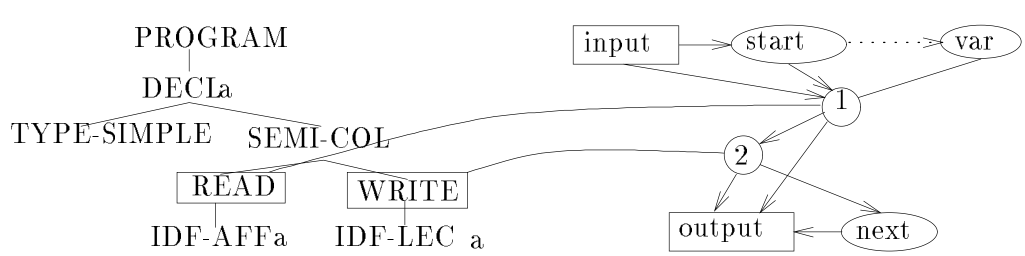

Figure C.4: Execution at step 4 of the macro program symbol `SEMI-COL`. This macro program symbol is used to compose two instructions. Cell 1 gives birth to another cell labeled 2. Cell 1 goes to read the macro program symbol that corresponds to the instruction `read a`. Cell 2 is blocked on the instruction `write a`

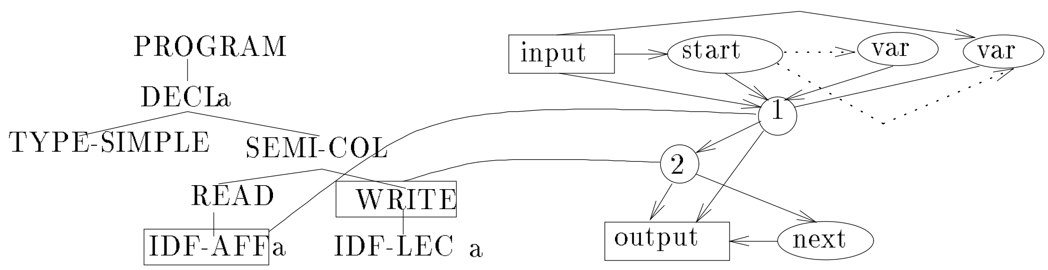

Figure C.5: Execution at step 5 of the macro program symbol `READ a` A neuron labeled `var` is created. It contains the new value of variable "a". This neuron is connected to the input pointer cell. it means that the value of "a" will be taken from the outside.

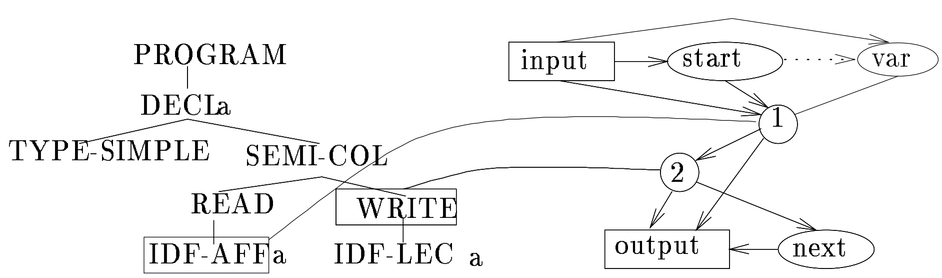

Figure C.6: Execution at step 6 of the macro program symbol `IDF-AFF`. The neuron "var" that contains the old value of "a" is deleted, because it has no more output links. The cell "2" is unblocked, it carries on its development.

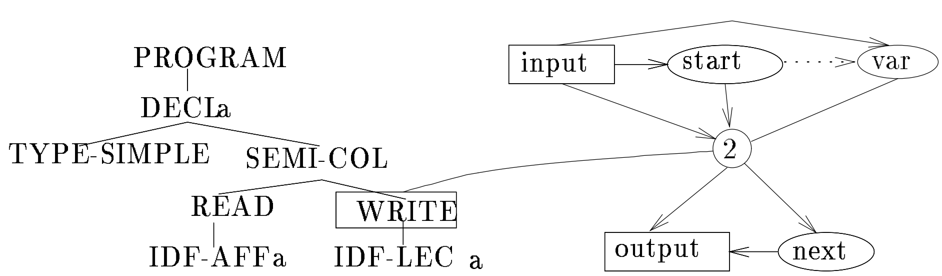

Figure C.7: Unblocking at step 7 of the cell "2". Before to start the compilation of the `WRITE`, cell "2" executes a small piece of cellular code. It merges the input links of cell "1". The cell "1" disappears. Now, the invariant is restored. Cell 2 can start the compilation of the next instruction.

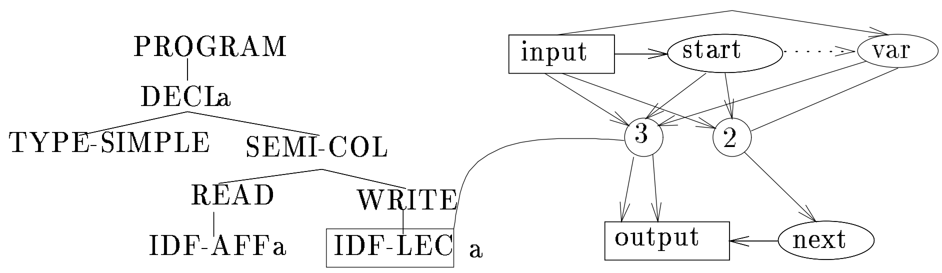

Figure C.8: Execution at step 8 of the macro program symbol `WRITE`. Cell 2 gives birth to another cell 3, connected to the output pointer cell. This cell 3 possesses all the input connections of cell 2. Cell 2 becomes a neuron.

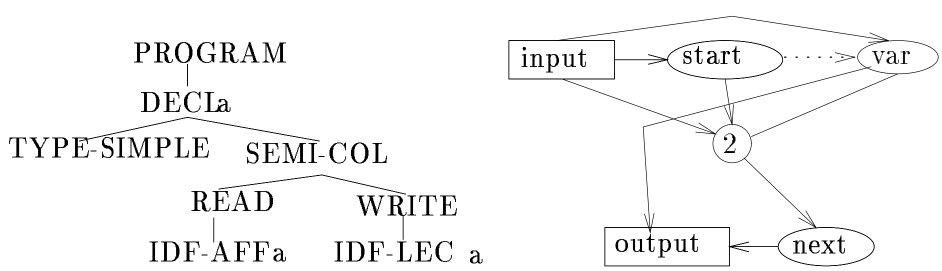

Figure C.9: Execution at step 9 of the macro program symbol `IDF-LECT`. Cell 3 select the input that corresponds to variable ”a”, and connect that input to the output pointer cell. Then, cell 3 disappears.

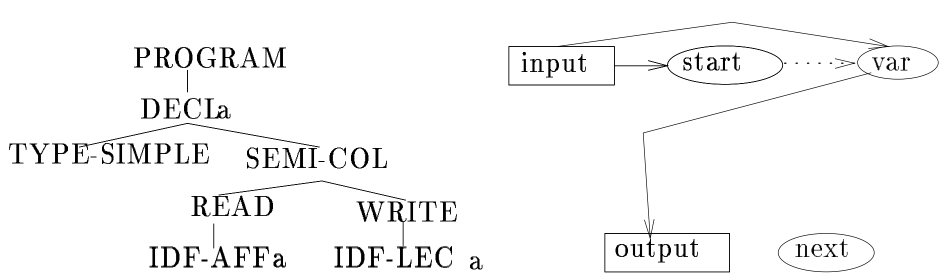

Figure C.10: The ”next” cell unblocks, and disappears. Its deletion leads to the suppression of neuron 2. The final neural net encompass two input neurons: the start cell and the input variable ”a”; and one output neuron: the variable ”a”. This neural net reads ”a” and write ”a”. It translates what is specified by the Pascal program.

C.2 The other macro program symbols

In this section we describe the macro program symbols that have not been illustrated in the preceding section, on the simple example of compilation.

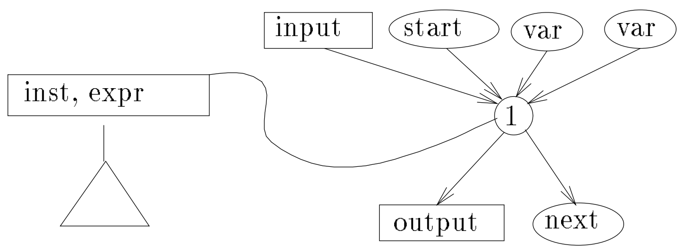

Figure C.11: Invariant pattern. We show here the network graph of cells after the declaration of two scalar variables. We can see clearly the two corresponding neurons labeled "var". These neurons "var" with the neuron "start" will constitute the first layer of the final neural network. The graph of this figure is an invariant pattern that will be taken for the initial configuration, for all the next macro program symbols. This avoids repetition and stresses the importance of the invariant pattern in the design of the macro program symbol. Cell 1 will be called the ancestor cell.

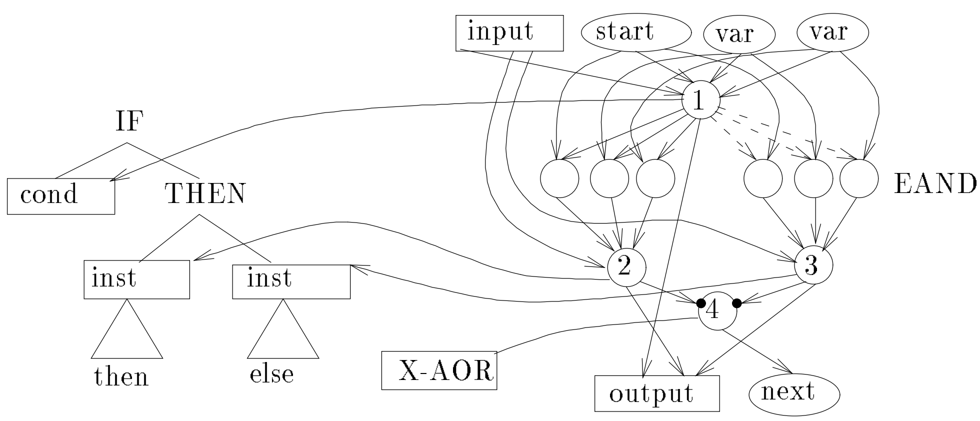

Figure C.12: Execution of the macro program symbols IF and THEN by the ancestor cell of the invariant pattern. For the sake of clarity, we have represented the combined action of IF and THEN. Four reading cells, and a layer of 6 neurons EAND are created. Cell 1 will develop a sub-neural net that allows to compute the value of the condition. TRUE is coded on value 1, and FALSE is coded on value -1. The logical operation NOT is realised using a weight -1 (dashed line). Depending on the value of the condition, the flow of values contained in the environment is sent to the left or to the right. The neurons EAND do that switching of the flow. The reading cell 2 and 3 develop respectively the body of the THEN and the body of the ELSE. Cell 4 possess two input sub-sites which are represented as small black disks. It is blocked on a piece of cellular code called X-AOR. It will gather the output lines coming out of the ELSE sub network and the THEN subnetwork.

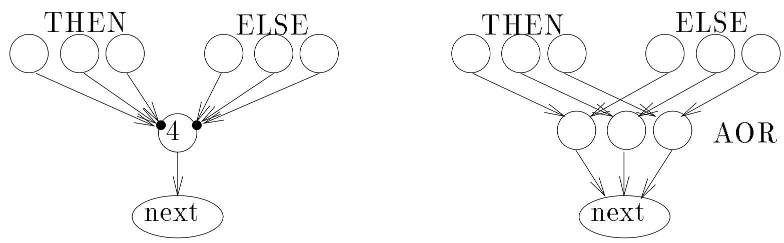

Figure C.13: Execution of the cellular code `X-AOR` by cell 4 of the preceding figure. Cell 4 has two input sub-sites that allows to distribute its input links into two lists. The cellular code allows to handle these two lists separately. It produces a layer of `AOR` neurons. This layer retrieves either the output environment from the body of the `THEN` or the output environment from the body of the `ELSE`; and transmits it to the next instruction.

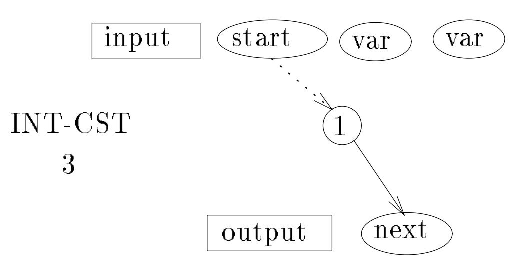

Figure C.14: Execution of the macro program symbol `INT-CST` by the ancestor cell of the invariant pattern. This macro program symbol has one parameter which is the value of the constant. Macro program symbol `INT-CST 3` has the following effect: It sets the bias of the ancestor cell to the value 3, and deletes all the links, except the link to the "start" neuron and the link to the "next" neuron. The sigmoid of the ancestor cell is set to the identity, and the ancestor cell becomes a neuron labeled 1. When the constant is needed, the neuron "start" is used to activate the neuron 1, that delivers the constant coded in its bias.

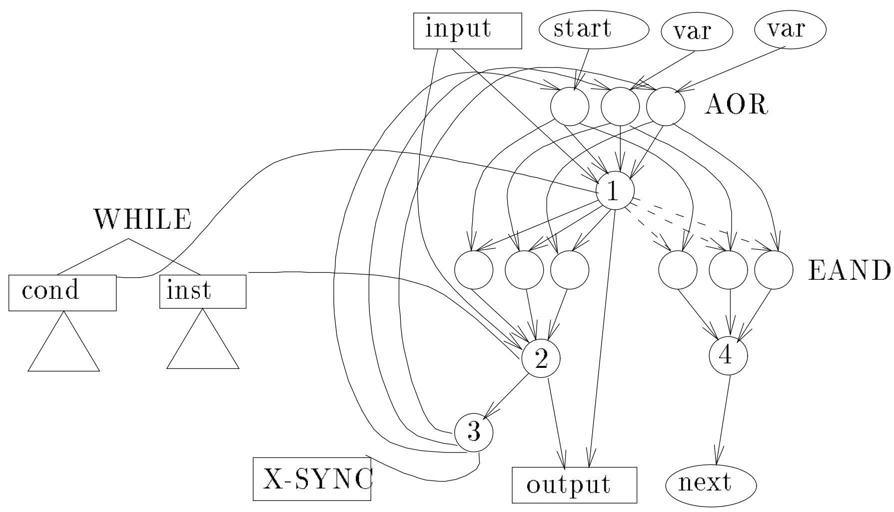

Figure C.15: Execution of the macro program symbol `WHILE` by the ancestor cell of the invariant pattern. Many cells are created: three reading cells, 3 `AOR` neurons, 6 `EAND` neurons, one neuron labeled 4, used to forward the environment. The layer of `AOR` neurons is used as a buffer that stores the environment between two iterations of the loop. The `AOR` dynamic is used because the flow of values either comes from the top of the network (first iteration) or from the bottom (other iterations), but it never comes from both the top and the bottom at the same time. The reading cell 1 develops a neural net that computes the condition of the while. Cell 2 develops the body of the `WHILE`. As in the case of the `IF`, the `EAND` neurons are used as a switching for the flow of activities. If the condition is true, the flow of values is forwarded in the body of the loop, else the flow of values turns to the right and exits the loop. Cell 3 develops an intermediate layer in the loop, which will synchronize the flow of activities, and will avoid collisions between activities.

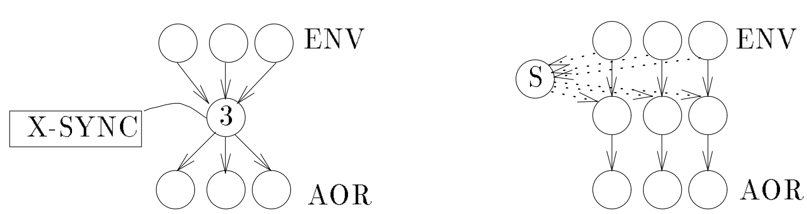

Figure C.16: Execution of the cellular code `X-SYNC` by cell 3 of the preceding figure. Cell 3 gives birth to a layer l of three neurons plus one neuron "S" whose dynamic is the normal dynamic. This layer is inserted between the layer of neuron that stores the output environment of the body of the `WHILE`, and the buffer layer of `AOR` at the beginning of the loop. The neuron "S" is connected from its input site, to all the neurons that contains the values of the environment. The neuron "S" is active only when all the environment is available. The neuron "S" is linked to the three neurons of layer l from its output site, with 0 weights. These connections are used to block the neurons while "S" is not active. So the three neurons of layer l are unblocked at the same time, and the flow of values is synchronized.

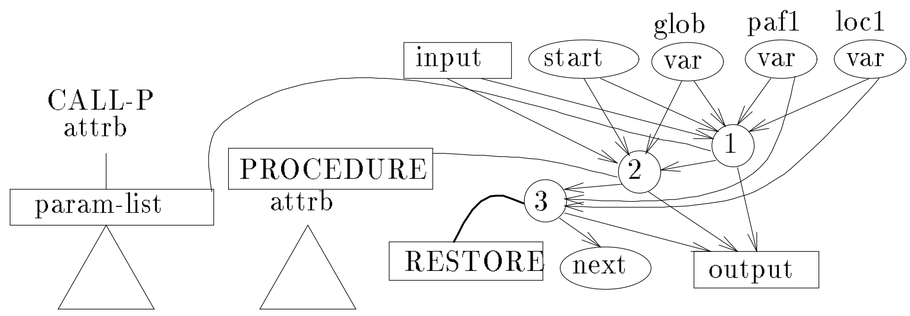

Figure C.17: Execution of the macro program symbol `CALL-P` by the ancestor cell of the invariant pattern. The environment has three variables. The `CALL-P` has one parameter (`attrb`) which is the name of the called procedure. Three cells are created. Cell 1 develops different sub-networks: one sub-network for each parameter to pass to the called procedure. Cell 2 develops the body of the called procedure. Its reading head is placed on the root of the parse tree that defines the called procedure. We stress that cell 2 is not connected to the neurons that contain the local variables of the calling procedure: `pef1` and `loc1`. These variables cannot be accessed from `proc2`. Cell 3 is blocked on a cellular code `RESTORE`. It will restore the environment of the calling procedure, when the translation of the called procedure is finished. Cell 3 keeps the connections to the local variables `pef1` and `loc1`.

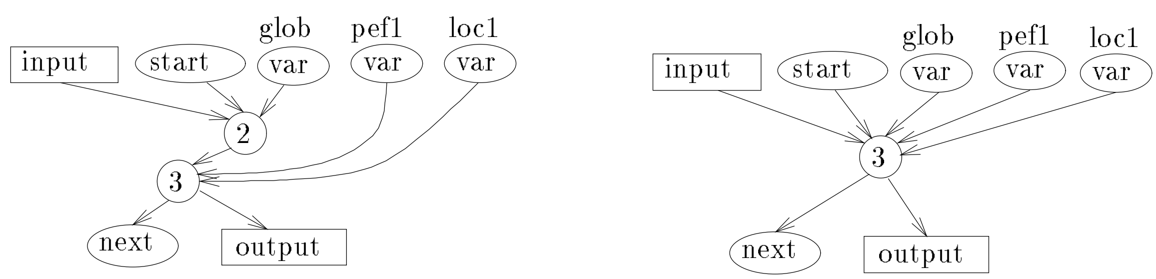

Figure C.18: Execution of the cellular code `RESTORE` by cell 3 of the preceding figure. When cell 3 is unblocked, the body of `proc2` is developed. Cell 3 is linked to cell 2 which is linked to the global variables, that may have been modified by the called procedure. The execution of `RESTORE` brings together the modified global variables, and the unchanged local variables of `proc1`.

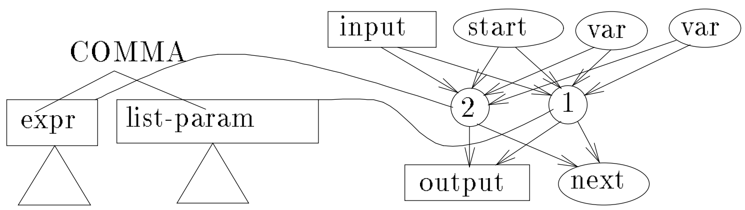

Figure C.19: Execution of the macro program symbol `COMMA` by the ancestor cell of the invariant pattern. This macro program symbol creates two cells. Cell 2 develops a sub-network for computing the value of a parameter, cell 1 continues to read the parameter list.

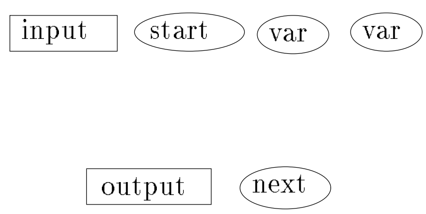

Figure C.20: Execution of the macro program symbol `NO-PARAM` by the ancestor cell of the invariant pattern. Assume that cell 1 of the preceding figure is ready to develop yet another parameter, although the end of the parameter list is reached. The end of the list is marked by a N̂O-PARAM that simply eliminates cell 1.

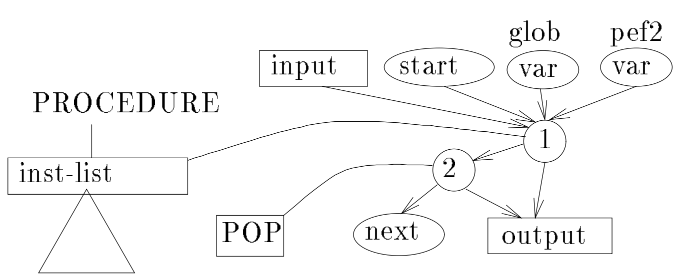

Figure C.21: Execution of the macro program symbol `PROCEDURE` by the ancestor cell of the invariant pattern. This macro program symbol adds an intermediate cell 2, blocked on a `POP` cellular code. This cell will be unblocked, when the body of the loop will be developed. Its role is to cut the connections to the local variables of the called procedure. Similarly, when a sub routine returns, the local variables are poped out from the stack. Cell 2 does not yet have the neuron `loc2` in its environment. This neuron is generated by the macro program symbols associated to the declaration of `proc2`.

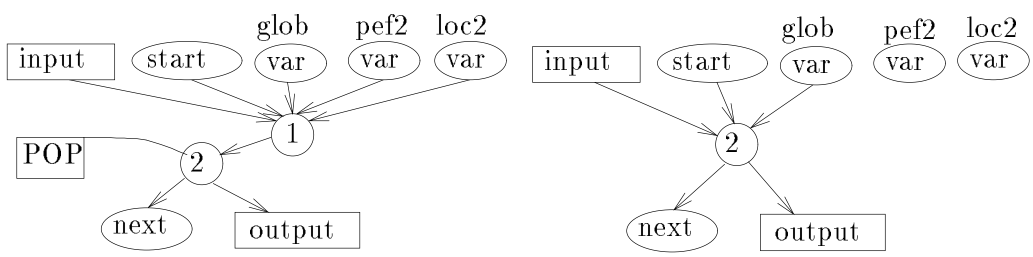

Figure C.22: Execution of the cellular code `POP` by cell 2 of the preceding figure. This macro program symbol allows to retrieve only that part of the environment which is global. The local variables are no longer connected to neuron 2 after the execution of `POP`. This macro program symbol is equivalent to the pop instruction of a machine language, used for the return of sub-routines.

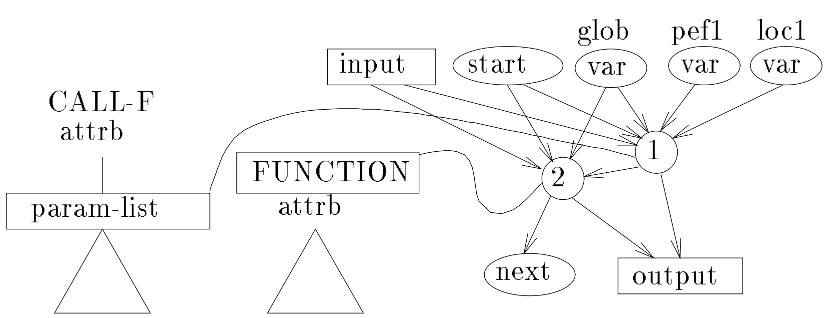

Figure C.23: Execution of the macro program symbol `CALL-F` by the ancestor cell of the invariant pattern. This macro program symbol has the same effect as `CALL-P` except that it does not create the cell 3, because it is not necessary to restore the environment.

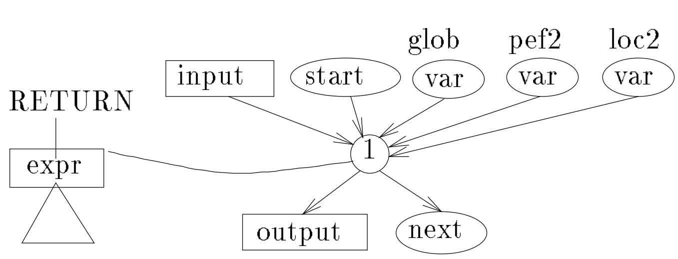

Figure C.24: Execution of the macro program symbol `RETURN` by the ancestor cell of the invariant pattern. This macro program symbol has a null effect.

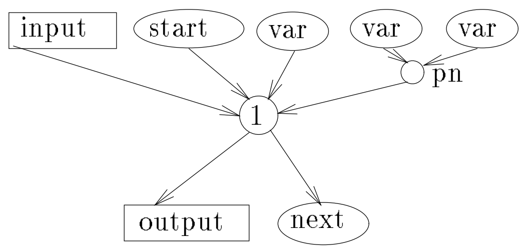

Figure C.25: Extended invariant pattern, the environment contains a scalar variable and a 1D array with two elements.

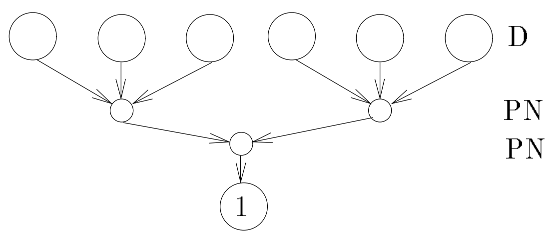

Figure C.26: Representation of a 2D array, using a neural tree of pointer neurons, of depth 2. Layers of pointer neurons are marked by "PN". The layer of neurons marked "D" contains the values of the array.

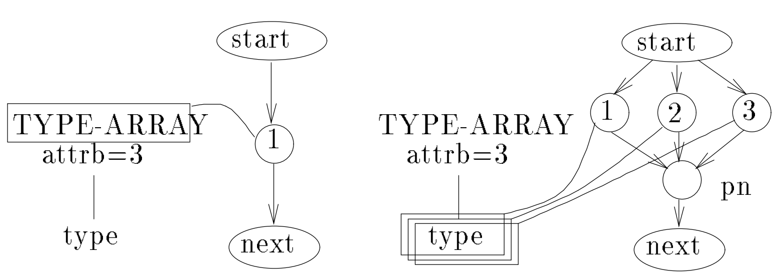

Figure C.27: Execution of the macro program symbol `TYPE ARRAY`. This macro program symbol allows to develop the neural tree of pointer neurons that is required to encode an array. We need d macro program symbols in order to represent an array of d dimensions. `TYPE-ARRAY` has one parameter which is the number of columns along the dimension corresponding to that macro program symbol. `TYPE-ARRAY` is the only macro program symbol where different cells go to read the same macro program symbol.

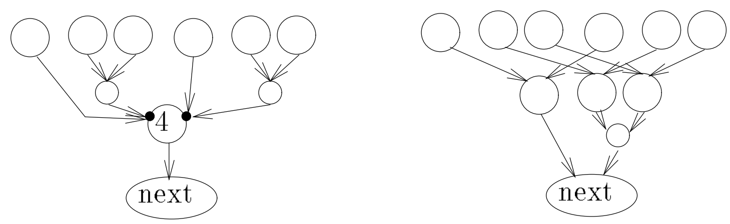

Figure C.28: Execution of the cellular code `X-AOR` by cell 4 of the figure that describes the `IF` macro program symbol. The result is the production of a layer of `AOR` neurons, and the duplication of the neural tree that gives a structure to the data. The layer of `AOR` neuron retrieves either the output environment from the body of the `THEN` or from the body of the `ELSE`, and transmits it to the next instruction.

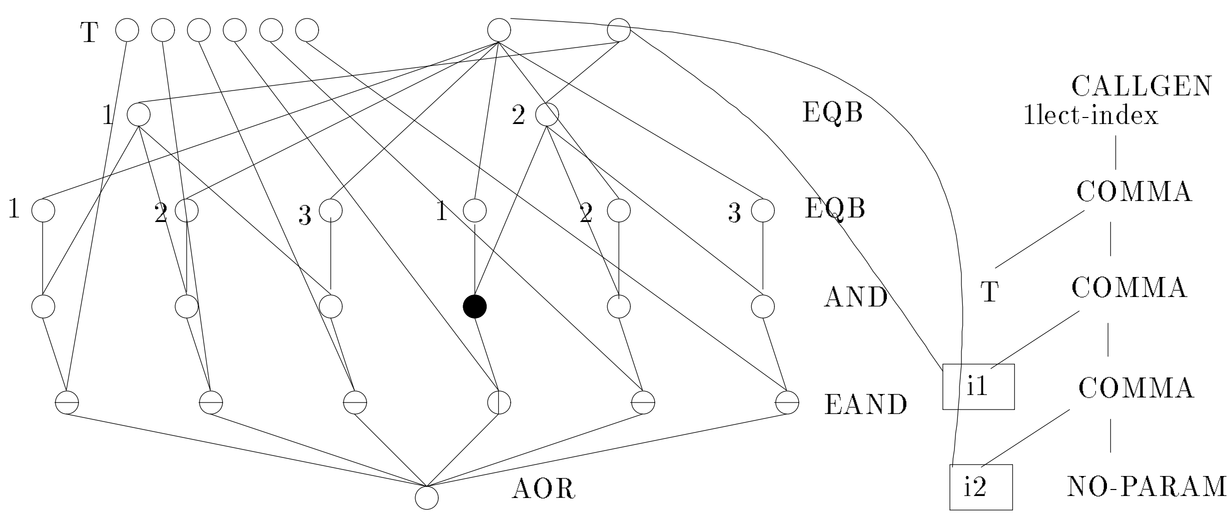

Figure C.29: Reading of a 2D array. The index `i1` is 2, the index `i2` is 1. The two first layers of `EQB`s have a sigmoid that test the equality to zero. They output 1 if the net input is 0, otherwise they output -1. These `EQB` neurons have different thresholds that are indicated. The third layer of neurons computes the logical `AND`s. In this layer, a single neuron outputs 1, the one that corresponds to the element that must be read. All the other neurons of this layer output -1. The fourth layer of neurons is a layer of `EAND` neurons. only one of these neuron is activated. It propagates the element of the array that is read. Last, a neuron `AOR` gathers all the output lines and retrieves the element that is read. The instruction `CALLGEN` is almost like `CALL-F`, except that all the links to the global variables are deleted. `CALLGEN` will be described at the end of this section

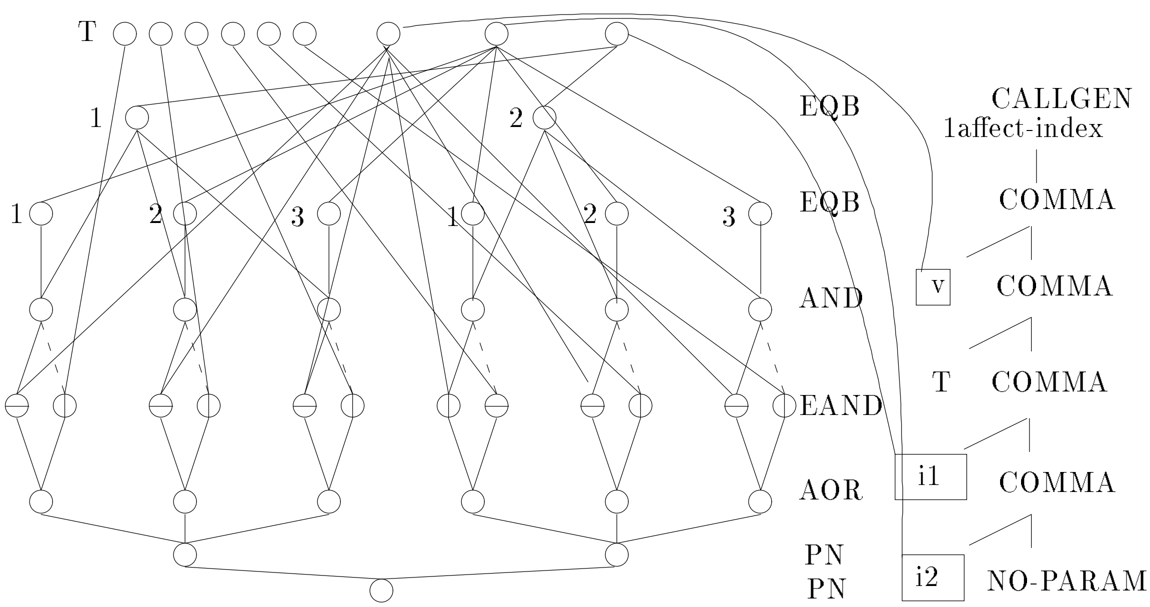

Figure C.30: Writing in a 2D array. The index `i1` is 2, the index `i2` is 1. The first three layers are the same as in the preceding figure. The fourth layer contains twice as much `EAND` neurons. These neurons `EAND` are distributed in pairs. Each pair corresponds to an element of the array. In each pair, one neuron is activated, and the other one is not activated. The left neuron is never activated except in a single case, when the element corresponds to the place where we want to write. The following layer is a layer of `AOR` neurons. Each `AOR` neuron corresponds to an element of the modified array. It retrieves the element of the old array, or the value v that is written if the element corresponds to the place where we are writing. Last, a neural tree of pointer neurons preserves the structure of 2D array.

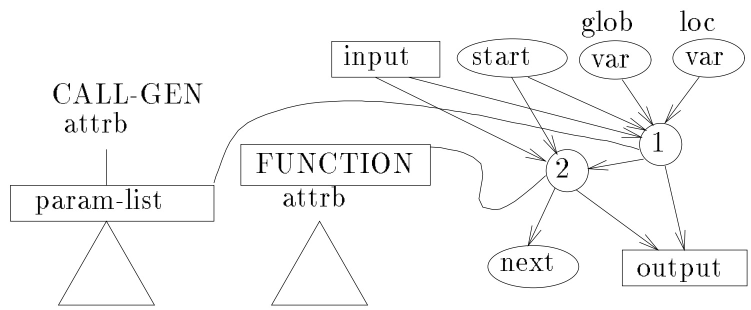

Figure C.31: Execution of the macro program symbol `CALLGEN` by the ancestor cell of the invariant pattern. `CALLGEN` has the same effect as `CALL-F` except that the cell number 2 which will execute the body of the function written in cellular code, does not have connections to the global variables of the Pascal program.

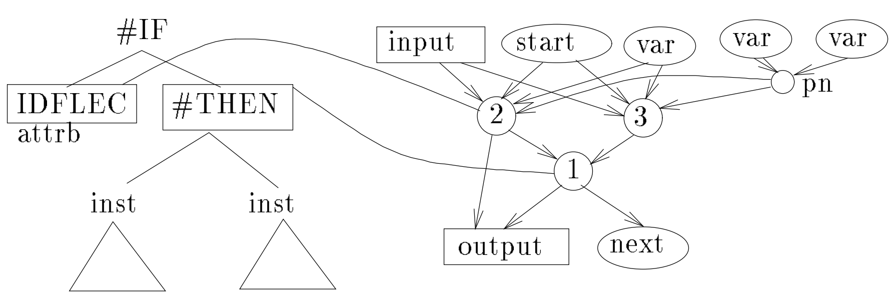

Figure C.32: Execution of the macro program symbol **#IF** by the ancestor cell of the invariant pattern. Three cells are created. Cell 2 goes to read a macro program symbol `IDF-LEC` that will select the array variable which is concerned by the test. Cell 3 is a neuron that will be used later by cell 1 to restore the environment, once the test is finished. Cell 1 will execute the next part of the condition encoded by the macro program symbol **#THEN**

C.3 Registers of the cell

Name	Role
Reading Head	Reads at a particular position on the cellular code
Life	Counts the number of recursive iterations
Bias	Stores the bias of the future neuron
Sigme	Stores the sigmoid
Dyn	Stores the dynamic
nu-lien	Points to a particular link
simplif	Specifies the level of simplifiability
x,y,dx,dy	Specifies wether the cell is a neuron or a reading cell

Table C.1: The registers of a cell

Moreover links have 3 registers, one for the weight `weight`, one for the state `state`, and one for the mark of beginning of a sub site `begin-sub-site`. Sites have a single register called `divisable`.

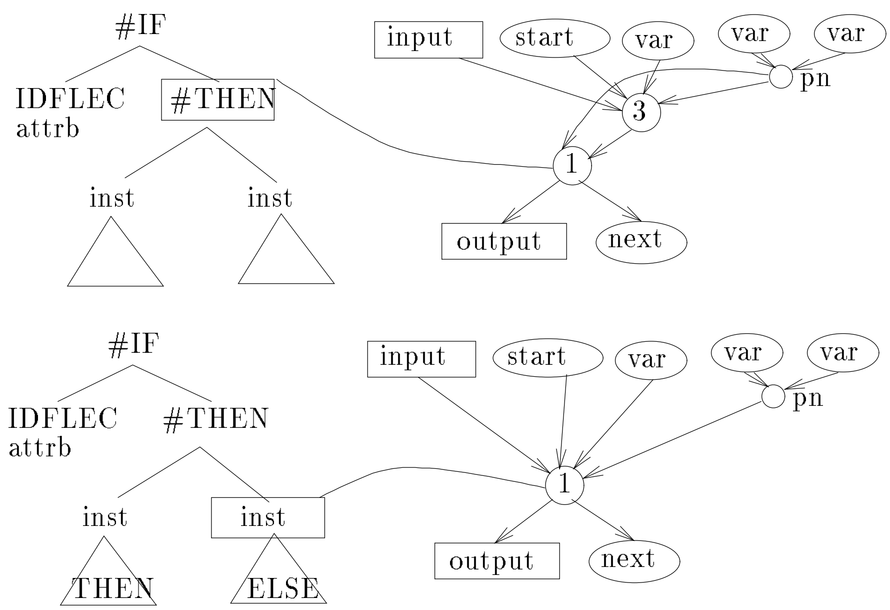

Figure C.33: Execution of the macro operator **`#THEN`** by cell number 1 of the preceding figure. After the execution of the macro program symbol `IDF-LEC`, cell 1 is unblocked and merges the links of its first input neighbor. The situation is described on top of the figure. The macro program symbol tests the number of input links of its first neighbor. Here this number is 2 because the array has two elements. If this number is greater than one, cell 1 places its reading head on the right subtree, as it is the case here. Otherwise it goes to read the left subtree. The left sub-tree contains the body of the **`#THEN`**, the right sub-tree contains the body of the **`#ELSE`**.

C.4 Syntax of the micro coding of cellular operators

The following grammar generates the microcode of operators

```
 <microcode> ::= <top> | <div> |  <reg> | <exe> | <aff>
       <top> ::= -TOP<segments> | TOP<segments>
       <div> ::= <div1> | <div2>
      <div1> ::= -DIV<segments> | DIV<segments>
  &lt;segments&gt; ::=&lt;segment&gt;&lt;segments&gt; | &lt;segment&gt;
   &lt;segment&gt; ::=<operator><operand> |<operator>^<operand> | s
 <operator> ::=  m | M | r | R | d | D
  <operand> ::= * | > | >= | < | <= | = | $ | #
      <div2> :: =HOM<number>
       <reg> ::= CELL<character> | SIT<character> | LNK<character>
       <exe> ::= EXE<character> | BRA<character>
       <aff> ::= AFF<character>
```

The non terminal <number> is rewritten into a number, The non terminal <character> is rewritten into a character. The characer is: For the non terminal CEL, SIT, LNK, the abbreviated name of a cell register, of a site or of a link; for EXE, it reminds what kind of movements the reading head does. For BRA, it specifies the kind of condition that is tested, for AFF is indicated the kind of the neuron that we got.

C.5 List of program symbols and their microcode

The letter x indicates that the program symbol uses an integer argument. i is the number of input links, o is the number of output links, r is the value of the link register.

```
******************Local Topological Transformation***************************
  CUTL x   TOPm-<m->    Cuts the input links i-x
  CUTRI x  TOPm<        Cuts the right input links starting from link x
  CUT x    TOPm<m>      Cuts input link  x
  CLIP     TOPm_<m_>    Cuts input link  r
  CUTO x   -TOPm<m>     Cuts output link  x
  MRG x    TOPm<d^*m>   Merges the input link of input neighbour x
  MG x     TOPm_<d_^*m_>Merges the input link of input neigbour r
  CPFO x  -TOPm<d^#m>=  Copies  first output of output cell x
  CHOPI x  TOPm=        Cuts all the input links except link x
  CHOPO x  -TOPm=       Cuts all the output links except link x
  CHHL     TOPm|>       Cuts the first input links from 1 until i/2
  CHHR     TOPm|<=      Cuts the last input links from i/2+1 until i.
  PUTF x   TOPm=m<m>    Puts link x in first position
  PUTL x   TOPm<m>m=    Puts link x in last position
  SWITCH x TOPm<m$m>$m= Makes a link permutation between  last link and link x
  KILL     TOP          Deletes the neurone
  CYC      TOPs         Creates a recurrent link
********Cell division into two childs which will execute a separate   code******
  PAR      DIVd*R*      Parallel  division
  SEQ      DIVs         Sequential division
  XSPL x   DIVm<=s      Gradual division
  SEP      DIVd*M|>     Separation division
  ADL x    DIVm<sm>     Adds a cell on link x
```

```
  AD  x    DIVm_<sm_>   Adds a cell on link r
  IPAR x   DIVm<sd=m>   Duplicates link x, Add a cell on link x
  SPLFI x  DIVd<=s      Duplicates first x inputs
  SPLFIL x DIVd-<=s     Duplicates first i-x inputs
  SPLLI x  DIVsd->=     Duplicates last input starting at link x
  SPLLIL x DIVsd>=      Duplicates last input starting at link i-x
******Cell division into more than two childs which will execute the same code***
  CLONE x  OLC          Clones into x childs, x is the argument
  SPLIT    HOM0         Splits into y childs, y is computed from the topology
  TAB      HOM1         Same has split, but child i has its bias set to i
  SPLITN   HOM2         Same as split, but the sub-sites are merged
************************Modification of  a register*****************************
  SBIAS x  CELb         Sets the bias to the argument
  SBIAS0   CELb 0       Sets the bias to 0
  SBIAS1   CELb 1       Sets the bias to 1
  INCBIAS  CEL+b 1      Increments the bias
  DECBIAS  CEL+b -1     Decrements the bias
  SSIGMO x CELs         Sets the type of sigmoid
  SDYN x   CELd         Sets the dymamic
  EVER x   CELe         Sets the level of simplifiability
  LR x     CELl         Sets the link register
  INCLR    CEL+l 1      Increments the link register
  DECLR    CEL+l -1     Decrements the link register
  SITE     SITD         Sets the divisability of the sub sites to the argument
  SITI     SITd         Sets the divisability of the input sub-sites to the argument
  SITO    -SITd        Sets the divisability of the output sub-sites to the argument
  DELSITE  SITO         Merges all the input sites and all the output sites
  DELSITI  SITo         Merges all the input sites
  DELSITO  -SITo        Merges all the output sites
  MULTSIT  SITm         Creates one input site for each input link
  VAL x    LNKv         Sets the value of the input weight r to x
  VALO x   -LNKv        Sets the value of the output weight r to x
  VAL+     LNKv 1       Sets the value of the input weight r to 1
  VAL-     LNKv -1      Sets the value of the input weight r to -1
  INC      LNK+v 1      Increments the value of the weight on the input link r
  DEC      LNK+v -1     Decrements the value of the weight on the input link r
  MULT2    LNK*v 2      Doubles the value of the weight on the input link r
  DIV2     LNK/v 2      Halves the value of the weight on the input link r
  RAND x   LNK?         Sets weight x to a random value in {-1, 0, 1}
  ON       LNKs 1       Sets the state of the link to on
  OFF      LNKs 0       Sets the state of the link to off
************************Managing of the execution order*************************
  WAIT x   EXEw         Waits x steps
  WAIT     EXEf         Waits one step if no arguments are supplied
  JMP  x   EXEj         Includes subnetwork x
  JMP      EXEj         Uses a pattern matching mechanism if no arguments are supplied
  REC      EXEr         Moves the reading head back to the root of the currently read subtree
  END      EXEe         Becomes a finished neuron
  BLOC     EXEq         Waits for the neigbour to loose their reading head
  NOP_0    EXEw         Is a label
  NOP_1    EXEw         Is a label
  DEF      EXEw         Starts a new branch that encodes a subnetwork
  BLIFE    BRAf         Tests if the value of the life register is one
  BLIF2    BRAg         Tests if the value of the life register is the initial value
  BUN x    BRAu         Tests if the neigbour has x input links
  BPN      BRAp         Tests if the neighbour is a pointer neuron
  BLR x    BRAl         Tests if the value of the link register equals the argument
************************Enhancing display of the graph***************************
```

```
  PN       AFFpn        Draws a pointer neuron
  VAR      AFFvar       Draws a neuron that contains the value of a variable
***********************Operators for labeling*********************************
  DINS     EXEf         It indicates insertion of the right sub-tree
  GINS     EXEf         It indicates insertion of the left sub-tree
  PN_      CELe         It is a pointer neuron
  VAR_     EXEe         It contains the initial value of a variable
  EAND_    EXEe         It has a dynamic EAND
  AOR_     EXEe         It has a dynamic AOR
  INF_     EXEe         It compares the 2 inputs
  SUP_     EXEe         It compares the 2 inputs
  INFEQ_   EXEe         It compares the 2 inputs
  SUPEQ_   EXEe         It compares the 2 inputs
  PLUS_    EXEe         It adds the 2 inputs
  MOINS_   EXEe         It substracts the first input from the second inpouts
  MULT_    EXEe         It multiplies the 2 inputs
  QUO_     EXEe         It divides the first input by the second inputs
  NOT_     EXEe         It does a logical NOT
  OR_      EXEe         It does a logical OR
  AND_     EXEe         It does a logical AND
  NEQ_     EXEe         It tests if the two inputs are different
  EQ_      EXEe         It tests if the two inputs are equal
  EQB_     EXEe         It tests if the input is equal to bias
  START_   EXEe         It is a Start neuron
```

C.6 Syntax of the parse trees that are used.

The following grammar generates parenthesized expressions that can be interpreted into correct parse trees. A parse tree can be the parse tree of the main program, the parse tree of a procedure, or the parse tree of a function. The compiler generates a list of trees, one tree for the main program, and one tree for each function or procedure. The parenthesized representation used here is a bit unusual. We do not put parenthesis when there is a single subtree; we only put a blanc. When there are two sub trees, we consider that the right sub tree is a trunc, and put only the left subtree between parenthesis. This special representation allows to write a simple grammar. Capital letters and parenthesis are terminals of the grammar. Non terminals are written using small letters, between brackets.

```
<parse_tree> ::= <program> | <procedure> | <function>
   <program> ::= PROGRAM <decl_list> <inst_list>
 <procedure> ::= PROCEDURE <decl_list> <inst_list>
  <function> ::= FUNCTION <decl_list> <inst_f>
 <decl_list> ::= DECL attrb (<type>)<decl_list> | DECL attrb <type>
      <type> ::= TYPE_ARRAY attrb <type> | TYPE_SIMPLE
 <inst_list> ::= SEMI_COLON (inst) <inst_list> | <inst>
    <inst_f> ::= SEMI_COLON (<inst>) <inst_f> |  RETURN <expr>
      <inst> ::= <write> | <read> | <assign> | <if> | <#if> |
                 <while> | <repeat> |<call_p> | <callgen>
    <assign> ::= ASSIGN ( <idf_aff> ) <expr>
      <read> ::= READ <idf_aff>
   <idf_aff> ::= IDF_AFF attrb
     <write> ::= WRITE <expr>
```

```
          <if> ::= IF(<expr>) <then>
        <then> ::= THEN (<inst_list>)<inst_list>
         <#if> ::= #IF(<idf_lect>) <#then>
       <#then> ::= #THEN (<inst_list>)<inst_list>
       <while> ::= WHILE (<expr>) <inst_list>
      <repeat> ::= REPEAT (<expr>) <inst_list>
      <call_p> ::= CALL_P attrb <param_list>
  <param_list> ::= COMA (<expr>)<param_list> | NO_PARAM
     <callgen> ::= CALLGEN attrb0 <param_list>
        <expr> ::= <read_array> | <write_array> | <idf_lect> |
                   <const> | <bin_op> | <un_op> | <call_f>
  <read_array> ::= CALLGEN attrb1  <param_list>
 <write_array> ::= CALLGEN attrb2 <param_list>
    <idf_lect> ::= IDF_LEC attrb
       <const> ::= INT_CST attrb
      <bin_op> ::= BINOP attrb (<expr>)<expr>
      <bin_op> ::= UNOP attrb <expr>
      <call_f> ::= CALL_F attrb <param_list>
```

This grammar generates parse trees with nodes having sometimes some attributes. We now indicates what each attribute represents.

Name	Function of the attribute
DECL	name of the declared variable
TYPE-ARRAY	dimension of the array
IDF-AFF	name of the variable to assign
IDF-LEC	name of the variable to read
INT-CST	value of the integer constant
CALL-P	name of the procedure to call
CALL-F	name of the function to call
CALL-GEN	attrb0 is the name of a pre-encoded solution
CALL-GEN	attrb1 is the name of a procedure for reading an array
CALL-GEN	attrb2 is the name of a procedure for writing an array
BINOP	name of the binary operator
UNOP	name of the unary operatot

Table C.2: The attributes of the parse tree

C.7 Predefinitions

These predefinitions are elementary cellular codes that are used many times. They are defined separately in order to reduce the size of the cellular code generated by the neural compiler.

C.7.1 Arithmetic operators.

Here we enumerate elementary codes that implement arithmetic operators, using neurons, as well as the AOR and EAND dynamic. The sigmoids are in increasing order: Identity(1), Pi-units(3), stair step-L(5), stair step-R(6), equality to zero (7) Div-unit (8)

OR OR SBIAS 2 (SSIGMO 5 *stair step-L* (OR-))

AND AND SSIGMO 5 *stair step -L* (AND-)

NEG NEG LR 1 (VAL -1 (NOT-))

INFEQ INFEQ LR 1 (VAL -1 (SSIGMO 6 *stair step-R* (INFEQ-)))

SUPEQ SUPEQ LR 2 (VAL -1 (SSIGMO 6 *stair step-R* (SUPEQ-)))

INF INF LR 1 (VAL -1 (SSIGMO 5 *stair step-L* (INF-)))

SUP SUP LR 2 (VAL -1 (SSIGMO 5 *stair step–L* (SUP-)))

EQ EQ LR 1 (VAL -1 (SSIGMO 7 *equality to zero* (EQ-)))

NEQ NEQ SEQ (JMP EQ) (JMP NEG)

PLUS PLUS SSIGMO 1 *identity* (PLUS-)

MOINS MOINS LR 2 (VAL -1 (SSIGMO 1 *identity* (MOINS-)))

MULT SSIGMO 3 *PI Unit* (MULT-)

QUO SSIGMO 8 *DIV Unit* (QUO-)

EOR AOR SDYN 2 (AOR-)

EAND EAND SDYN 3 (EAND-)

C.7.2 Reading and writing in an array

We present here cellular code of function that allow to read and write in a multi dimensinal array. Each of this function needs three elementary piece of code that recurively call themselves.

1AFFECT-INDEX SITE 1 (CUTLE 3 (CUTO 1 (MULTSIT (SEQ (ADL 1 (SITE 0 (JMP CREAT-EQB)) (MRG -2 (SPLIT (JMP 1AFFECT-SUIT)))) (SITE 0 (BLOC (PN- 2)))))))

CREAT-EQB TAB (VAL -1 (SSIGMO 7 (EQB-)))

1AFFECT-SUIT BUN 3 (JMP 2AFFECT-SUIT) (SEQ (SPLLIL 3 (SITE 0 (CUTRI 3 (ADL 2 (JMP CREAT-EQB) (SPLIT (JMP AND))))) (MRG -2 (SPLIT (JMP 1AFFECT-SUIT)))) (SITE 0 (BLOC (PN- 2))))

2AFFECT-SUIT BPN 2 (SITE 0 (JMP 3AFFECT-SUIT)) (MRG 3 (MRG 2 (SEQ (SPLIT (JMP 2AFFECT-SUIT)) (SITE 0 (BLOC (PN- 2))))))

3AFFECT-SUIT SEQ (PAR (CUT 3 (LR 1 (VAL -1 (JMP EAND)))) (CUT 2 (JMP EAND))) (JMP AOR)

1INDEX-LEC SITE 1 (CUTLE 3 (CUTO 1 (MULTSIT (SEQ (ADL 1 (SITE 0 (JMP CREAT-EQB)) (MRG -1 (SPLIT (JMP 1INDEX-SUIT)))) (BLOC (JMP 2INDEX-SUIT))))))

1INDEX-SUIT BUN 2 (JMP SUPER-EAND) (SPLLIL 3 (SITE 0 (CUTRI 3 (ADL 2 (JMP CREAT-EQB) (SPLIT (JMP AND))))) (MRG -1 (SPLIT (JMP 1INDEX-SUIT))))

2INDEX-SUIT SPLIT (BPN 1 (JMP AOR) (SEQ (LR -1 (JMP SUPER-MRG)) (SITE 0 (BLOC (PN- 2)))))

C.7.3 Recursive macro program symbol for the manipulatin of the environmnet

We enumerate here elementary codes that are used in many places in the cellular code generated by the compiler. These code apply recursively the same operation to all the environment.

the macro SUPER-MRG BLR 0 (MG (ADDLR -1 (JMP SUPER-MRG))) (JMP 2INDEX-SUIT)

the macro SUPER-EAND SUPER-EAND SPLIT (BPN 2 (JMP EAND) (MRG 2 (SEQ (JMP SUPER-EAND) (SITE 0 (BLOC (PN- 2))))))

the macro SUPER-AOR SUPER-AOR SEQ (MRG 2 (MRG 1 (SPLIT (BPN 1 (JMP AOR) (JMP SUPER-AOR))))) (SITE 0 (BLOC (PN- 2)))

the macro SUPER-DYN SUPER-DYN SPLIT (BPN 1 (EVER 1 (SDYN 2)) (SEQ (MRG 1 (JMP SUPER-DYN)) (SITE 0 (BLOC (PN- 2)))))

the macro SYNCHRO SYNCHRO IPAR 1 (VAL 0 (EVER 0)) (SITE 0 (BLOC (DELSITO (SPLITN (EVER 1)))))

the macro REPEAND REPEAND SPLIT (BPN 2 (JMP EAND) (MRG 2 (JMP REPEAND)))

the macro WHILEND WHILEND SPLIT (BPN 1 (END) (MRG 1 (JMP WHILEND)))

the macro READ-SUIT READ-SUIT BPN 3 (CUT 3 (LR 2 (VAL 0))) (SEQ (MRG 3 (SPLIT (JMP READ-SUIT))) (SITE 0 (BLOC (PN- 2))))

the macro WRITE-SUIT WRITE-SUIT BPN 1 (END) (MRG 1 (SPLIT (JMP WRITE-SUIT)))

C.8 Rules for rewriting nodes of the parse tree

There is one rules for each non terminal of grammar C.6.

IF SITE 1 (SPLLIL 1 (CPFO 1 (SITE 0 (DELSIT (RINS)))) (SEQ (SITE 0 (SPLFI 2 (CUT 2 (PAR (WAIT 2 (SITE 1 (JMP SUPER-EAND))) (LR 1 *NOT SIMPLIFIE* (VAL -1 (SITE 1 (JMP SUPER-EAND)))))) (CUT 1 (CPFO 1 (LINS))))) (BLOC (CUTO 1 (JMP SUPER-AOR)))))

THEN PAR (CUT 3 (BLOC (DELSIT (RINS)))) (CUT 2 (BLOC (DELSIT (LINS))))

REPEAT SITE 1 (CYC (SITE 0 (XSPL 1 (SITE 1 (JMP SUPER-DYN)) (SPLFI 1 (CPFO 1 (BLOC (DELSIT (RINS)))) (BLOC (MRG 2 (SITE 1 (SPLLIL 2 (CPFO 1 (SITE 0 (DELSIT (LINS)))) (PAR (CUTO 1 (CUTO 1 (LR 1 (VAL -1 (SEQ (WAIT 2 (JMP REPEAND)) (JMP SYNCHRO)))))) (CHOPO 2 (SEQ (JMP SUPER-EAND) (SITE 0 (BLOC (END))))))))))))))

WHILE SITE 1 (CYC (SITE 0 (XSPL 1 (SITE 1 (JMP SUPER-DYN)) (BLOC (SITE 1 (SPLLIL 1 (CPFO 1 (SITE 0 (DELSIT (LINS)))) (PAR (CUTO 2 (SPLFI 2 (CUT 2 (JMP SUPER-EAND)) (SITE 0 (CUT 1 (BLOC (SEQ (CPFO 1 (RINS)) (BLOC (SITE 1 (CUTO 1 (SEQ (MRG 1 (WAIT (JMP WHILEND))) (JMP SYNCHRO)))))))))) (CHOPO 2 (LR 1 (VAL -1 (SEQ (CUT 2 (JMP SUPER-EAND)) (SITE 0 (BLOC (END))))))))))))))

PROGRAM IPAR 1 (START-) (SEQ (CPFO 1 (SWITCH 1 (LINS))) (BLOC (MRG 1 (CUTO 1))))

NO-INST CUTO 1 (CUT 1)

DECL IPAR 2 (RINS) (PUTL 2 (BLOC (LINS)))

TYPE-ARRAY SEQ (CLONE ATTR (LINS)) (BLOC (EVER 2 (PN)))

TYPE-SIMPLE VAR (LR 1 (VAL 0 (EVER 2)))

SEMICOL SPLFI 1 (CPFO 1 (LINS)) (BLOC (MRG 2 (RINS)))

ASSIGN SPLFIL 1 (CPFO 1 (RINS)) (BLOC (LINS))

IDF-AFF SWITCH attr (CUTL 1 (CUT 1 (CUTO 1)))

IDF-LEC CUTO 1 (CHOP attr (BPN 1 (END) (MRG 1 (PN- 2))))

COMMA PAR (LINS) (RINS)

UNOP SEQ (CPFO 1 (LINS)) (CUTO 1 (JMP attr))

BINOP SEQ (CPFO 1 (PAR (LINS) (RINS))) (CUTO 1 (JMP ATTR))

INT-CST CHOP 2 (CUTO 1 (SBIAS ATTR (LR 1 (VAL 0 (EVER 2)))))

CALL-F SPLFI GLOBAL (CPFO 1 (RINS)) (BLOC (JMP ATTR (CUTO 1 (CHOP 2 (LR 1 (VAL 0))))))

NO-PARAM KILL ()

FUNCTION ou RETURN LINS ()

CALL-P SPLLI LOCAL (CPFO 1 (SPLFI GLOBAL (CPFO 1 (RINS)) (BLOC (JMP ATTR (END))))) (BLOC (MRG 1 (CUTO 1)))

PROCEDURE SEQ (CPFO 1 (LINS)) (BLOC (MRG 1 (CUTRI GLOBAL (CUTO 1))))

READ SPLFIL 1 (SPLFI 2 (CHOP ATTR) (MRG 3 (SITE 1 (MULTSIT (JMP READ-SUIT))))) (BLOC (RINS))

WRITE PAR (CHOPO 1 (SEQ (CPFO 1 (RINS)) (BLOC (JMP WRITE-SUIT])))) (CUT 1 (CUTO 1))

#IF SEQ (CPFO 1 (PAR (CUTO 1) (RINS))) (BLOC (MRG 2 (LINS)))

#THEN BUN ATTR (CHOP 1 (MRG 1 (RINS))) (CHOP 1 (MRG 1 (LINS)))

CALLGEN SPLFI 2 (CPFO 1 (RINS)) (BLOC (JMP ATTR (CUTO 1 (CHOP 2 (LR 1 (VAL 0))))))

C.9 Library of functions for handling arrays.

gauche CUTLE 3 (CUTO 1 (MRG 1 (CHHR 1 (PN- 2))))

droite CUTLE 3 (CUTO 1 (MRG 1 (CHHL 1 (PN- 2))))

concat CUTLE 3 (CUTO 1 (MRG 2 (MRG 1 (PN- 2))))

int-to-array CUTLE 3 (CUTO 1 (PN- 2))

array-to-int CUTLE 3 (CUTO 1 (MRG 1 (BPN 1 (END) (MRG 1 (PN- 2)))))

randomise CUTLE 3 (CUTO 1 (JMP rand))

rand BPN 1 (LR 1 (VAL 0 (RAND))) (SEQ (MRG 1 (SPLIT (JMP rand))) (PN- 2))

C.10 Trained neural networks

The following cellular code correspond to neural networks that are included in the pascal program animal.p

retine CUTLE 3 (CUTO 1 (SEQ (MRG 1 (SPLIT)) (SEQ (CLO 10 (EVER 0)) (BLOC (PN- 2)))))

position-object CUTLE 3 (CUTO 1 (SEQ (SEQ (MRG 1 (CLO 2 (EVER 0))) (CLO 2 (EVER 0))) (EVER 0)))

predator CUTLE 3 (CUTO 1 (SEQ (SEQ (MRG 1 (CLO 3 (EVER 0))) (CLO 3 (EVER 0))) (EVER 0)))

motor CUTLE 3 (CUTO 1 (SEQ (EVER 0) (SEQ (CLO 2 (EVER 0)) (SEQ (CLO 4 (EVER 0)) (PN- 2)))))

C.11 The extended switch learning

```
Let c be the neuron currently processed;
oni:=the old net input of c;
For each neuron n that fans into c through link l
      If (l's weight is not null)
            Try to modify n's activity;
      nni = the new net input of c; d=Abs(nni-oni)/2;
      Switch(Comp(nni,oni))
      Case 0 : if(Small(oni)) r_n += r_c;
               else r_n += r_c * p1
               w_n += w_n + w_c;
      Case + : r_n += w_c * p2;
      Case - : w_n += w_c * p3 * d; r_n += r_c * p4 * d;
  If (n is an input unit) P5=1; Else P5=1/p5;
  If (n's activity is not null)
    For all v in {-1,0,1}, v not equal to l's initial weight
```

```
      Try to set l's weight to v;
      nni = the new net input of c; d=Abs(nni-oni)/2;
      Switch(Comp(nni,oni))
      Case 0 : if (Small(oni)) r_l[v] += r_c * P5;
               else r_l[v] += r_c * p1 * P5
               w_l[v] += w_c;
      Case + : r_l[v] += w_c * p2 * P5;
      Case - : w_l[v] += w_c * p3 * d * P5; r_l[v] += r_c * p4 * d * P5;
  Elseif n is not input unit
    For all v in {-1,0,1}, v not equal to l's initial weight
      Try to both set l's weight to v and modifyn's activity;
      nni = the new net input of c; d=Abs(nni-oni)/2;
      Switch(Comp(nni,oni))
      Case 0 : if(Small(oni)) r_n += r_c * p6; r_l[v] += r_c * P5 * p6;
           else r_n += r_c * p1 * p6; r_l[v] += r_c * p1 * P5 * p6;
           w_n += w_c * p6; w_l[v] += w_c * p6;
      Case + : r_n += w_c * p2 * p6; r_l[v] += w_c * p2 * P5 * p6;
      Case - : w_n += w_c * p3 * d * p6; w_l[v] += w_c * p3 * d * P5 * p6;
               r_n += r_c * p4 * d * p6; r_l[v] += r_c * p4 * d * P5 * p6;
```

This algorithm describes how to computes the value r_n, w_n, $w_l[v]$, $r_l[v]$, $v \in \{-1, 0, 1\}$, starting from the output unit and going backwards to the input units, as in backpropagation. the subscript n refers to the unit that is currently processed. and the subscript l refers to a link. These values are then used to select which weight to modify, and the new value in which to modify it. Function `Comp` returns 0 if the net input change of sign, else it returns the sign of the difference between the absolute value of its first and second argument. Function `Small` returns 1 if its argument is 0 or 1, else it returns 0.The operator `+=` increments the variable of the left side by the quantity in the right side. The positive integers $p_i, i = 1..6$ parametrize the learning algorithm.