Efficient Computer Morphogenesis: A Pictorial Demonstration

Frédéric Gruau

SFI WORKING PAPER: 1994-04-027

SANTA FE INSTITUTE

Efficient Computer Morphogenesis: a Pictorial Demonstration

Frédéric Gruau

Centre d'Etude Nucléaire de Grenoble
Département de Recherche Fondamentale Matiére Condensée
17 rue des Martyrs, 38041 Grenoble
gruau@lip.ens-lyon.fr

Ecole Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme
46 Allée d'Italie
69364 Lyon Cedex 07, France

April 29, 1994

Abstract

This paper illustrates that artificial morphogenesis can be a computationaly efficient technique. Artificial morphogenesis can develop graph grammar into modular Artificial Neural Networks (ANN), made of a combination of more simple sub networks. The genetic algorithm is used to evolve coded grammar that generate ANNs for a simplified six-legged robot. The genetic algorithm can automatically decompose a problem into sub problems, generate a sub-ANN for solving the sub-problem, and instanciate copies of this sub-ANN to build a higher level ANN that solves the problem. We support our argumentation with pictures describing the morphogenesis and illustrate how ANN structures are evolved.

1 Introduction

An Artificial Neural Networks (ANN) is a graph of simple computing elements called units, which are an abstract model of the biological neuron. There have recently been some attempts to get inspiration from biological morphogenesis, and to apply it to encode ANN [Gruau 1992], [Parisi and Nolfi 1992], [Belew 1993]. The idea is to indirectly represent an ANN, and to use an evolutionary algorithm to evolve high level representations for computational problem solving or artificial life simulations. Instead of directly describing a graph data structure like a list of connections from unit to unit, artificial morphogenesis describes how to build the neural net by applying rules of cell division. Starting with a single cell, artificial morphogenesis develops a graph of cells using repeated applications of the rules. When the development is finished, a cell is mapped to an ANN's unit, and the graph of cells can be interpreted as an ANN. The system of rules is modeled as a formal grammar, and the different approaches can be classified depending on the particular kind of formal grammars involved.

Artificial morphogenesis can be traced back to [Mjolness, Sharp and Alpert 1988] and [Kitano 1990] who proposed the first examples of evolving formal grammars in this context. Nevertheless, although the idea of high level representation based on a grammar can be found in their work, their implementations do not use cell division. Both Mjolness & all and Kitano use matrix grammars, that is, they develop matrix families instead of developing a graph of cells. This has many drawbacks, as was discussed in [Gruau 1992].

More recently, Parisi and Nolfi proposed geometric grammars. The object that undergoes division is a point in a two-dimensional space. The growth of neuron's axon is modeled but not

the cell division process. Another approach proposed by Belew simulates cell division using a two dimensional cellular automaton. Belew proposes a model with a context dependent grammar, where the way a cell divides is influenced by the neighbor cells. These two approaches are more targeted at modeling biology rather than exploiting morphogenesis for computational problem solving. Like Kitano and Mjolness, the problems they solve are trivial, or at least, they have already been solved by other methods. My interest is to see how ideas from biological morphogenesis can be exploited to generate ANNs that solve difficult problems. We proposed a system for artificial morphogenesis called cellular encoding in [Gruau 1992] based on cell division, where a cell is just the node of a graph. Cellular encoding encodes a graph grammar. We have implemented a neural compiler called JaNNeT [Gruau 1993b] that compiles a Pascal program into the cellular code of an ANN that simulates the Pascal program. JaNNeT demonstrates the expressive power of cellular encoding. We have proved several other theoretical properties of cellular encoding in [Gruau 1994 a]. If one wants to use artificial morphogenesis for problem solving, we think it is important to prove theoretical properties of the underlying encoding rather than just doing computer simulations. This gives a hint whether the evolutionary algorithm is going to be successful at exploring the space of codes and whether complex problems can be solved or interesting behavior observed.

I claim that artificial morphogenesis can be used to solve complex problems. It can generate regular neural networks that exploit the regularity of the problem to solve. In [Gruau 1992], using cellular encoding, the evolutionary algorithm was able to generate recursive graph grammars (or cellular codes) that develop families of arbitrarily large neural networks for computing the Boolean functions parity, symmetry, and decoder of arbitrary large number of inputs. To our knowledge, these particular problems have never been solved with any other methods. Cellular encoding allows to control the recursive development in a precise way, and to stop it after exactly n loops through the rules of the recursive grammar. The ANN number n that computes a Boolean function with n inputs is made of n copies of the same sub network, (or 2^n in the case of the decoder). The cellular code specifies in a homogeneous way the sub network and how to put together copies of it.

I have often been criticized that cellular encoding could only generate neural networks for regular Boolean functions. It is true that cellular encoding especially fits Boolean functions, because it is possible for such functions to define regular ANN architectures, using simple units with ± 1 weights, and Boolean activities. I think these functions are an ideal initial benchmark for artificial morphogenesis experiments because the optimal architectures are known, and one can make comparison. My claim is that cellular encoding is efficient whenever the problem to solve has a certain amount of regularity and can be decomposed into a hierarchy of sub problems. I believe that most problems have a lot of regularities, not only Boolean functions. In this paper, the method is used to solve a more realistic problem. I perform the genetic synthesis of an ANN for locomotion of a simplified six-legged robot. This problem has a regularity that can be exploited by artificial morphogenesis: Beer and Gallagher (1993) proposed a model of ANN made of six copies of the same sub network, each of which controls one leg. The connections between the sub networks are also regular. Using these symmetries, Beer and Gallagher were able to collapse the genetic code into a 200 bit string and to perform genetic synthesis of the neural network. In this paper I solve a slightly simpler version of the same problem, but on the other hand, I do not help the evolutionary algorithm by using my knowledge about the symmetries. Instead, artificial morphogenesis makes it possible to automatically find symmetries. The evolutionary algorithm decomposes the problem into sub problems, generates a sub network for solving the sub problem, and produces copies of this sub network to build a higher level ANN that solves the problem. The only fitness measure

we use is the distance covered by the animal.

2 Cellular Encoding revisited

Cellular encoding is a method for encoding neural networks. In this paper we present an updated and improved version of cellular encoding compared to the one in [Gruau 1994b]. Cellular encoding uses a very abstract notion of cells. A cell has an input site and an output site. It is linked to other cells, with directed and ordered links that fan into the cell at the input site and fan out from the cell at the output site. A cell also possesses a list of internal registers that represent a local memory and store labels. The data structure of an ANN is a directed labeled graph. My cell concept was made simplest as possible to capture exactly, and nothing more than what is needed to describe a directed and labeled graph. It is inspired from computer science instead of biology. The cellular code is based on local graph transformations or graph rewriting rules that act upon cells. Furthermore, there is a growing number of scientists working on graph grammars [Graph Grammar proceedings 1990] who have shown that graph grammars are very powerful at specifying complex objects, compared to more traditional string grammars, or even tree grammars and L-systems.

Examples of possible graph transformations used in cellular encoding are represented in Figure 1. Picture (a) represents an initial graph of cells. It is composed of one central cell connected to 6 input neighbors and 8 output neighbors. The remaining pictures, (b) to (l), show the effect of different graph transformations acting upon the central cell. The graph transformations can be classified into cell divisions, local topology transformations, and modifications of weights.

- Picture (b) to (h) represent the effect of different cell divisions. A cell division replaces one cell called the mother cell by two cells called child cells. One can imagine many possible cell divisions depending on the way the links of the mother cell are inherited by the child cells. The links are ordered and it is possible to refer to a link by using its number. In picture (a) the number of the links are represented. A sub list of consecutive links is specified by the number of the first link and the number of the last link in the sub list. A particular cell division is implemented by copying one or many sublists of links, from the mother cell to each of the child cell. A cell division must also specifies whether the two child cells will be linked or not. For practical purposes, we give a one-letter name to the graph transformation, and the set of letters will be the set of alleles used with the genetic algorithm. The particular letter we use does not have a particular meaning. Division "S" represented in picture (b) is the sequential division. In the sequential division, the first child cell inherits the input links, the second child cell inherits the output links and the first child cell is connected to the second cell. Division "P" represented in picture (c) is the parallel division. Both child cells inherit both the input and output links from the mother cell. Hence, each link is duplicated. The child cells are not connected. Divisions "S" and "P" are canonical divisions, because they are the most simple we could think of: all the other divisions do not handle all the links in a uniform way, independently from their position. They are not given a name like parallel or sequential. Division "T" is like "S", except that the input links number one and the output link number one are duplicated. Picture (e) and (f) look similar. In fact they represent two possible effects of the same cell division "A". Division "G" and "H" are symmetric to one another with respect to the inputs and outputs. They allow a cell to be inserted on the output or the input link number one.

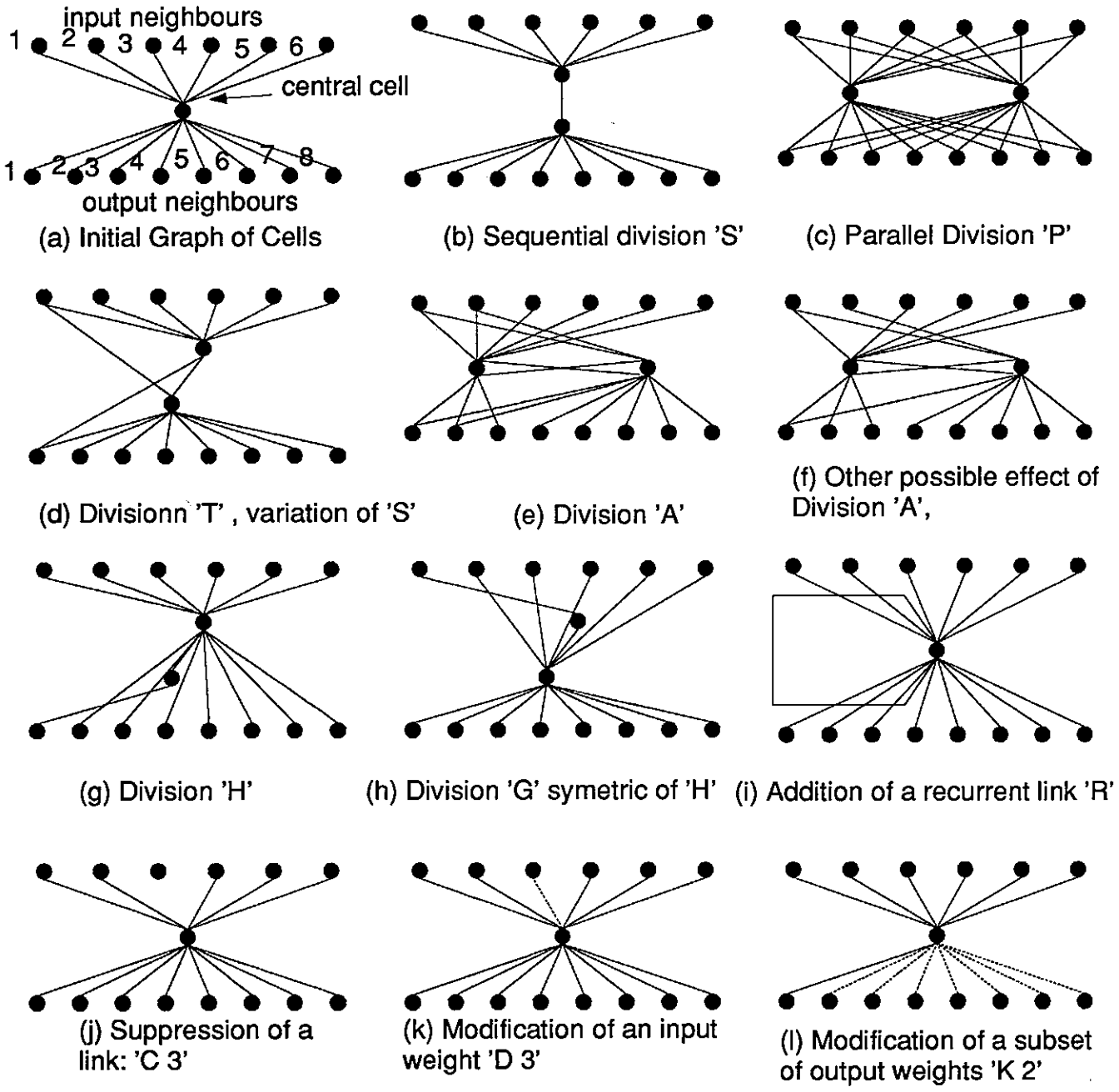

Figure 1: examples of local graph transformations

- Picture (i) and (j) represent graph transformations that locally transform the topology. The first one called "R" adds a recurrent link to the cell, the second one called "C" has an argument 3 — it cuts link number 3.

- The remaining two pictures describe graph transformations that have an argument, and modify the weights. The first is "D", the value of the argument is 3: it sets the input link number 3 to -1. The second is "K" with argument 2: it sets all the output links starting at 2 to -1.

Once more, the name of all these transformations do not carry a particular meaning. The development consists in applying successive graph transformations on a graph of cells and makes it grow into an ANN. In order to combine many graph transformations into a cellular code we used an ordered list of labeled trees instead of a set of grammatical rules. An example of cellular code is represented in Figure 2. The trees are labeled by names of graph transformation and are called grammar trees. The reader must not confuse grammar trees and tree grammars. A gammar tree refers to a grammar encoded as a tree, whereas a tree grammar refers to a grammar that rewrites trees. Each cell carries a duplicate copy of the cellular code (i.e., the set of grammar trees) and has an internal register called reading head that points to a particular position of the grammar tree. At each step of the development, each cell executes the graph transformation pointed to by its reading head.

Program symbols indicating cell division are labeling nodes of arity two, so that when a cell divides, the first child cell goes to read the left sub tree, and the second child cell goes to read the right sub tree. Program symbols indicating other graph transformations are labeling nodes of arity one, and once the transformation is executed, the cell will simply moves its reading head on the unique sub tree. Cells can execute other instructions than graph transformations: they can execute instruction for cell differentiation into an ANN unit, and instructions for piloting the reading head. All the possible instructions together are called program symbols. Since at each step of the development, all the reading heads move one level down in the tree; they come to reach the leaves. When a reading head reads the leaf of a tree, two possibilities may happen.

- It can encounters a program symbol such as "U" or "L" in figure 2. These two program symbols differentiate the cell into an ANN unit, with distinct particular features. Here, for example, U and L specify different sigmoids. ANN unit are cells that have terminated their development and lost their reading-head.

- It can read a program symbol "n" which has an argument d. If the number of the tree that is currently read is x, the cell moves its reading head on the root of the grammar tree $n+x$. For example, the program symbol "n 1" moves the reading head on the root of the next grammar tree. The program symbol "n 0" backtracks the reading head to the root of the grammar tree that is currently read, the program symbol "n 2" jumps on the next-to-the-next grammar tree. This is a reference mechanism using relative addresses.

During a step of the development, the cells execute their program symbols one after the other. The order 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. The waiting program symbol denoted "W" has no effect: it makes the cell wait for its next rewriting step. It is necessary for those cases where the development process must be controlled by generating appropriate delays.

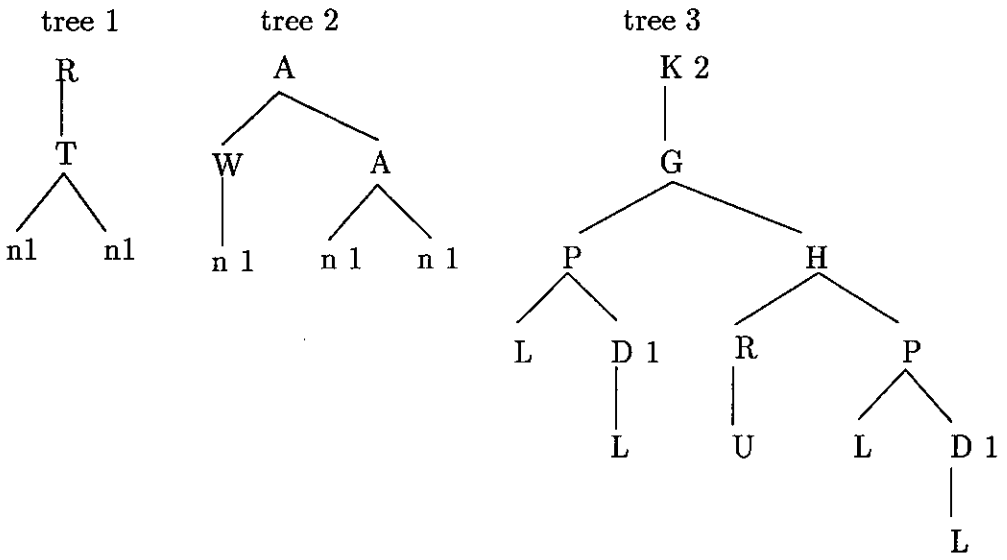

Figure 2: An example of cellular code designed by hand.

A simplified problem of six-legged robot locomotion is described in Section 3. Consider the problem of finding an ANN for this locomotion problem. The input units are sensory inputs that test the position of the legs. The output units commands the legs. The model of ANN unit is described in more details in section 4. Units have integer thresholds, and connections are weighted -1 or $+1$. We use continuous noisy units in which activity is updated with a differential equation and ranges between -2048 and $+2048$. The value 2048 was chosen to put sufficiently smoothness in the system. The cellular code presented in Figure 2 has been designed by hand. It encodes an ANN that solves the six legged locomotion problem. This neural net has been successfully tested for all the 64 possible initial position of the legs, where each leg is either full forward or full backward. The design is made of six oscillators coupled by inhibitory connections. It is inspired by the Pearson model described in [Beer 1990]. In Figures 3 and 4 we show the development of the cellular code presented in Figure 2. Each cell is represented by a circle. Inside the circle, we write the name (one letter) of the program symbol currently read by the reading head. This letter is very small but can still be read.

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. The pointer cells are represented by two parallel vertical lines. In the first two pictures of figure 3, they can be confused with the link that connects them because this link is also vertical. The pointer cells become apparent in the remaining pictures. The input pointer cell is always located at the top middle of each picture, and the output pointer cell

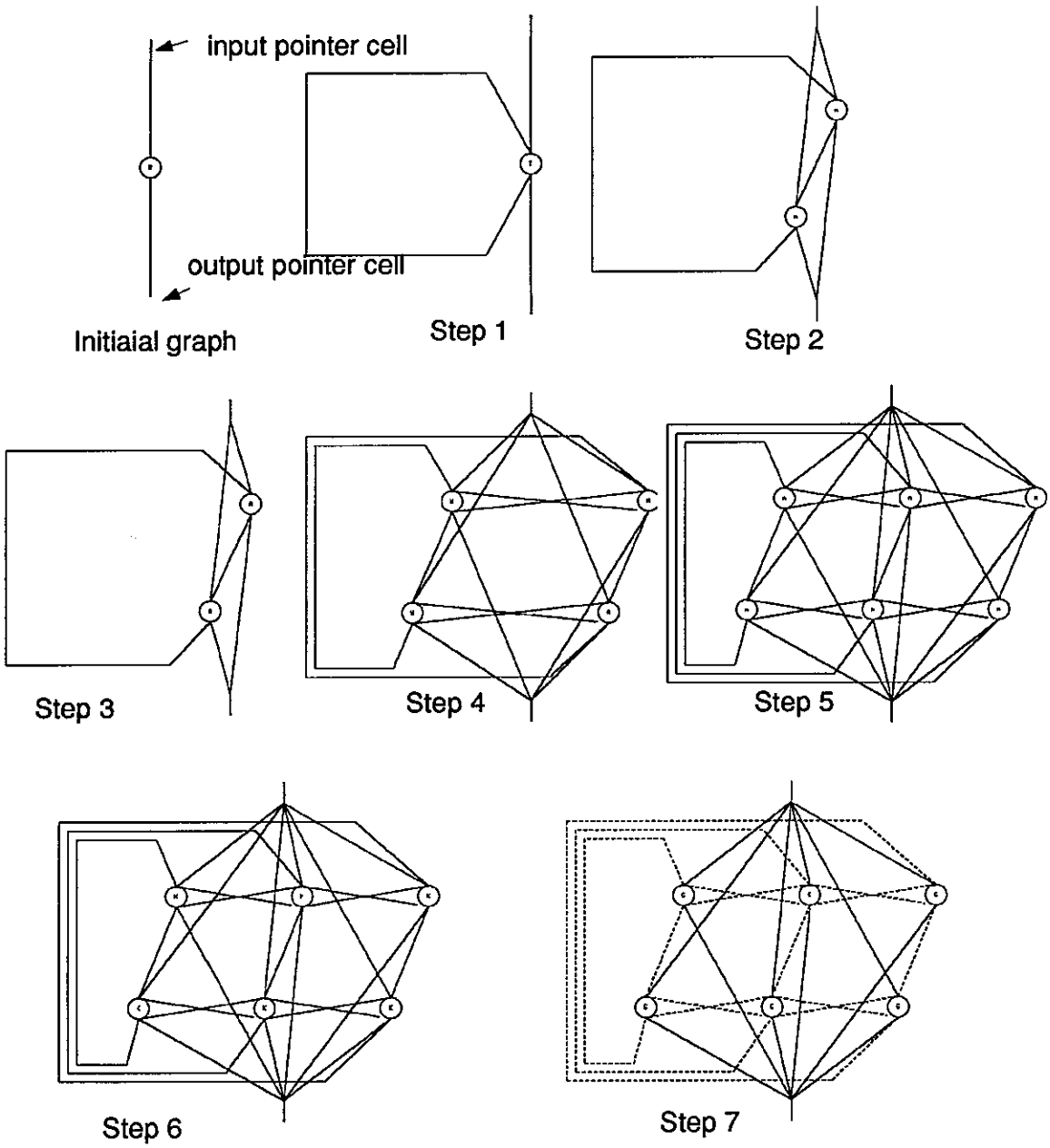

Figure 3: The development of a hand made cellular code, the first 7 steps

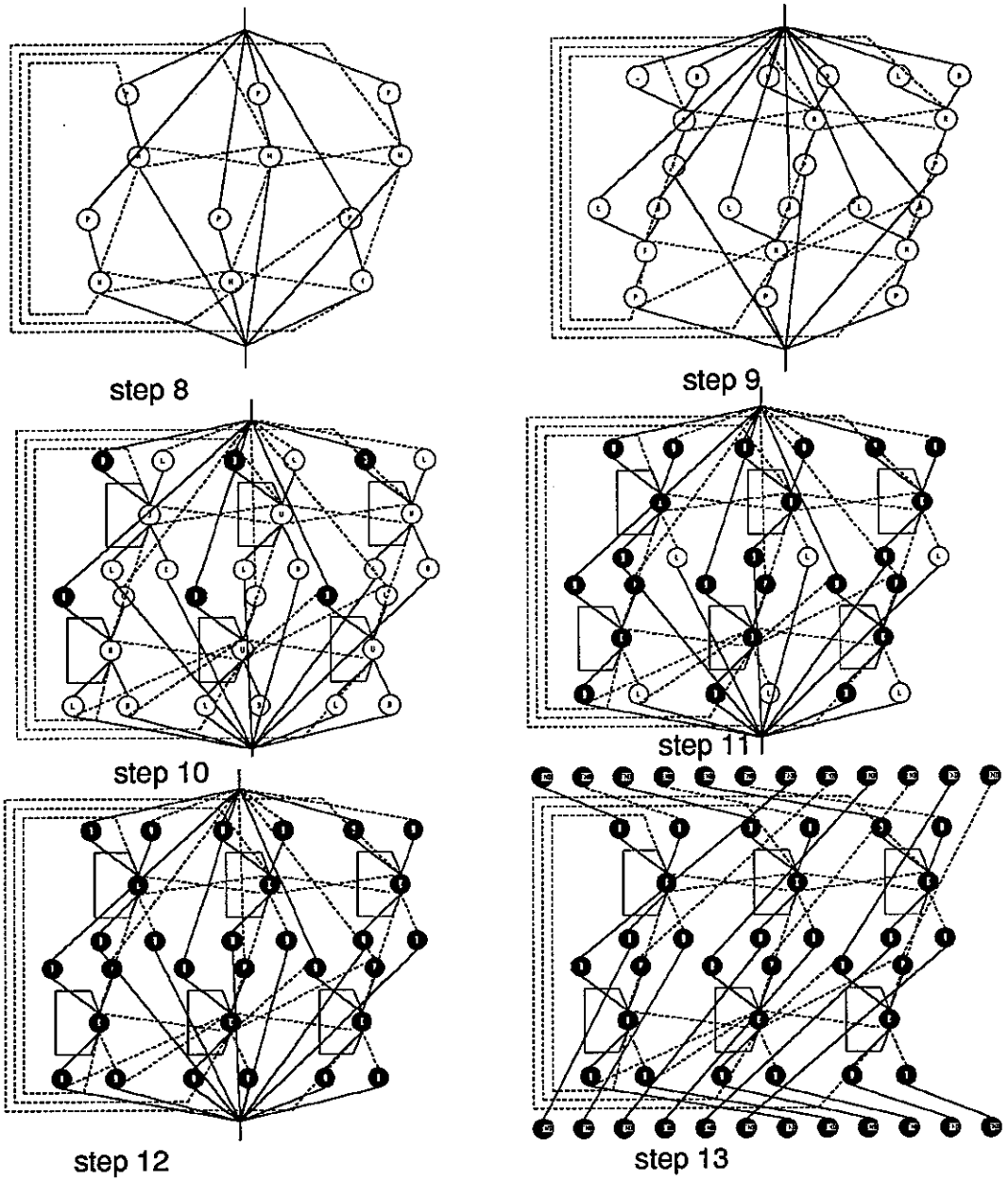

Figure 4: last steps of the development of the hand made cellular code

is always located at the bottom middle. Consider the picture at the top left of Figure 3. Initially, the reading head of the ancestor cell is located on the root of tree 1, and reads the program symbol "R". 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 units of an ANN. and make up the neural network. The input and output pointer cells to which the ancestor is linked do not execute any program symbol. Rather, at the end of the developmental process, the upper input pointer cell is connected to the set of input units, while the lower output 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 and are replaced by duplicate input and output units, as shown in the picture in the lower right corner of Figure 4.

3 A benchmark problem

We use the model of artificial six-legged insect described in [Beer 1990] with one difference: we did not used a motor neuron to control the feet. The aim of this simplification is to save computer time in the experiments. We think this simplified problem is a good benchmark for artificial morphogenesis because it is nontrivial, more realistic than boolean function, and it has an internal regularity. The problem can be stated abstractly as: "Generate a system of six coupled oscillators, such that whatever the initial conditions, the system will always move into the same global attractor, where the oscillators are divided in two classes. They are in phase within a class, but antiphase between classes, and their amplitude is tuned to a particular value".

We now describe the embodied version of the problem. A leg controller has two output neurons, one that controls the return stroke (RS), another one that controls the power stroke (PS). If RS is activated, the leg is pulled back with speed proportional to RS's activity. If PS is activated, the leg exerts a force on the body proportional to PS's activity. By convention, if both RS and PS are activated, RS "wins" and PS is ignored. Each foot has an internal state. A foot can be either up or down. A leg can begin the return stroke only if the foot is up, and it can begin a power stroke only if the foot is down. In either case, if the foot is not in the correct position, it takes one time step to put it in the right position. This takes into account the inertia of the leg and induces a selective pressure towards using the full range of the leg angle. If the feet had no state there would be a trivial solution to the problem, namely to alternate power stroke and return stroke at each time step. The rear legs have a 10% greater angle of movement, in order to break symmetry effects (this has been used by Beer). Due to friction, a dragging leg exerts a constant force pushing backward, proportional to the speed of the animal. At each time step, we sum the forces exerted by the legs which are down, and the negative forces, plus another global friction force pushing backward, proportional to the speed of the robot. This sum is used to update the speed of the robot. If the center of mass of the robot lies outside the polygon formed by the feet which are down, the robot falls down and its speed drops to zero. Otherwise the speed is used to update the position of the robot, and the joint angle of the legs which are down. We choose all the parameters of the model so that the tripod gait would be the one that enabled the robot to cover the maximum distance. This gait seems the most difficult to find over the range of possible regular gaits. In order to help coordination of the legs, each leg controller has two input units: Anterior Extrem Position (AEP) and Posterior Extreme Position (PEP). AEP 's activity is maximum if the leg is at its anterior extreme position and 0 otherwise. PEP 's activity is maximum if the leg is at its posterior extreme

position and 0 otherwise.

4 The continuous, noisy, neural model

To solve the problem of locomotion of a six-legged robot the ANN must be able to store an internal state. Hence, it needs recurrent links. We selected a particular neural model based on trial and error. We tried to build a neural net that solved the problem before performing optimization with the genetic algorithm. We began with the simplest model (from an implementation point of view). Everything was discrete, the weights were ± 1 and the activities where in $\{-1, 0, 1\}$. The sigmoids were piecewise linear functions, and the thresholds were integers. The activities of the neuron were updated one after the other, in a fixed order. With this simple model, we were able to design a solution by hand. But the GA would produce solutions very sensitive to the order in which activities are updated. For a particular predetermined order the network would oscillate correctly. But it we started to update the activities in a different order the ANN moved only four legs. So we switch to the parallel updating which made it difficult to build a solution by hand. In the parallel dynamic, the activities of all the neurons are updated at the same time. We studied this dynamic on the ANN developed in Figures 3 and 4. With the parallel dynamic, this ANN falls into a wrong attractor for a particular subset of initial positions of the leg, where it uses only four legs instead of six. There are 64 possible initial positions of the leg, because each leg is initialized either at its anterior extreme position or its posterior extreme position.

We were able to decrease the size of the basin of attraction to eight items, by using a continuous time update of the neurons activities, as advocated in [Beer and Gallagher 1993]. The activity of a neuron i is updated according to $\tau * (a_i(t + \delta t) - a_i(t)) = \delta t * s_\alpha(netinput_i)$. The net input is the weighted sum of the neighbor's activities minus the threshold of the neuron, τ_i is a time constant whose value is 3 in our experiments. The activities are now integers that ranges from -2048 to $+2048$. We updated the neuron activities three times before updating the body state. We further reduced the number of unsuccessful initial leg positions to four items by using an asymmetric sigmoid. We call s_α the sigmoid parametrized by α which takes values $-\alpha/2$ at infinite negative value, $+\alpha$ at infinite positive value, and crosses the origin. Finally, we added random noise in the model. Each time a unit computes its activity, it adds a random number uniformly distributed between -10 and $+10$. Noise was able to make the wrong attractor vanish completely. This positive effect of noise seemed magic. Noise probably helps doing a better exploration of the ANN's internal states. During the rhythmic activity, when the legs switch from RS to PS or vice versa, the ANN's state must come very near to the boarder between the wrong attractor and the right attractor. A little noise is then enough to slip from the wrong attractor into the right attractor.

5 The random learning method

We apply a learning method on each ANN produced by the evolutionnary algorithm, in order to speed up the genetic search. We randomly choose a weight w, and modify it to a random value in $\{0, 1, -1\}$. We then modify the cellular code of that ANN so that the modified cellular code produces the same ANN where w is modified. This is called back coding. Assume we are using genome that are sets of more than one grammar trees, and the back coding modifies the grammar tree of a sub network. In this case the back coding has an interesting side effect: wherever this sub

network will be instanciated in the final ANN, the weight modification will be reproduced.

We compute the performance of the ANN developed with the modified code. We compare that performance with the performance before learning. If the performance increases, we accept the weight change. If not, we still accept the weight change with a probability $e^{-0.1}$. We use a single epoch because learning is expensive. In this context learning nearly doubles the time of fitness evaluation.

There are two possible ways to exploit the information produced by learning. One is to have learning modifies the fitness function and forget the back coded information from one generation to the other. This is called the Baldwin effect. The modification of the fitness is done implicitely, because the learning method increases the fitness. Baldwin proposed that the effect of learning could make the fitness landscape more easy to climb. The alternative way is to transmit the learned information to the offspring. This mechanism has been proposed by Lamarck. Although it not biologically plausible Lamarckian learning can be used in a computer. Past results [Gruau and Whitley 1993] have shown that the Baldwin effect can speed up the genetic search in a sometime more efficient way than lamarckian learning. In this paper we used Lamarckian learning because the learning method was random and therefore seldom increased the fitness. In this context it is intuitively more usefull to remenber a successfull weight change when that happens, because it is not likely to be reproduced easily.

Back-coding was done using program-symbols "M", "N", "F", which respectively have the same effect as "D", "I", "C": set the weight number n to, respectively, $-1, +1, 0$. Here n is the argument of the program symbol. We used a different name in order to keep track of the information that has been learnt and inherited by the Lamarckian strategy.

6 The use of 32 different fitnesses

The fitness of a given ANN for the problem of the six legged locomotion robot is the average of the distance covered by the robot on a certain sample of initial leg positions. Since the number of input units and output units is also evolved, it might not match the number of inputs and outputs of the problem. If there are too many input our output units, we simply set their activity to 0, and ignore them. If there are not enough, we input only part of the input vector, and filled the undefined components of the output vector with 0.

Our aim is not merely to produce an ANN for six legged locomotion, we also want our ANN to be general. Whatever the initial condition of the six legs, we want our robot to perform the tripod gait after a transient of reasonable length. So in all of our experiments, we carefully test the neural networks produced by the Genetic Algorithm (GA) for the 64 possible initial positions of the leg, where each of the legs is initially either at its anterior extreme position, or posterior extreme position. This tests whether the neural net can generalize. We point out that in [Beer and Gallagher 1993] no test for generalization are reported.

One of the most delicate tasks was to devise a good fitness that measures robustness, together with a good termination criterion. By "good" we mean a fitness that is not too computationally expensive, and such that when the fitness reaches the level where we assume the solution is acceptable and stop the genetic algorithm, then the solution generalizes over the 64 possible initial positions. We cannot test all the initial leg positions for each neural network produced by the GA. This would be too time consuming. The test of a single initial position is done for 150 time steps, this means that our ANNs, which can contain up to 120 neurons, will undergo 150 iterations of the

parallel dynamic for a single initial position of the legs. For the particular ANNs generated by the GA, it turns out to take between 0.3 and 0.5 seconds on either a Sparc 10 or an I860 processor. For each neural network, we do one epoch of random learning. The initial position of the legs must be tested a second time. Hence the cost of fitness evaluation using a single initial position lies between 0.6 and 1 seconds.

We tried successively four different fitness functions differing in the way the initial leg positions are chosen. The termination criterion was whether the GA had found an ANN walking a greater distance than the hand coded solution, It is a common feature of all the fitnesses. First we ran the GA using a single initial position of the legs that we though was the most difficult: all the legs are at AEP. The GA found solutions that did well for this particular initial position, but for many other initial positions the robot was paralyzed or used only four legs. In order to provide a fitness for more robusts ANNs synthesis, we tested the ANN on three selected initial positions instead of one. The run time was three times as long. Yet, the solution found by the GA did not generalize correctly either. The fitness used for the next experiment was more elaborate. For each evaluation, we pick a random initial position, and if the neural network is able to walk past a threshold distance, then we pick another random initial position. A neural network may be evaluated up to 10 initial random positions, but to proceed to the next one, it must have been able to walk the threshold distance on the preceding one. So time is saved, because very often, the evaluation will stop at the first initial random position. As individuals become better, more time is spent to evaluate them. The threshold distance was chosen as the distance walked by the hand coded solution, multiplied by 0.9. The termination criteria was to generate a neural network having the same performance of the hand coded solution on the 10 patterns. Yet the solution found by the GA did not generalize either. What happened is that the GA generated from time to time sequences of initial positions that were easy, or matched the ANN being evaluated; a particular ANN did well enough on this sequence; the termination criterion was fulfiled and the GA stopped. So instead of searching robust ANNs, the GA was searching easy sequences of initial leg positions.

We used another idea to defeat this problem. We divided the population into 32 sub populations, and gave a different fitness to each sub population. This is easy to implement, because we use a parallel GA that runs on a 32 processor parallel machine. We carefully selected a fixed set of seven difficult initial positions. A given fitness was built as follows: choose a particular (not random) initial position in the set; if the distance walked exceeds the threshold, choose another particular initial position, and so on, until all the set has been tried. Each sub population chooses initial positions in a particular sequence. For a given sub population, the sequence stays fixed during all the runs. During the first generation, each sub population concentrates on one of the seven initial positions. Hence, there are seven different fitness functions. When the network walked passed the threshold distance, the sub population begin to learn neural networks that can walk starting from two different initial positions. At this stage there are $7 * 6/2 = 21$ fitness functions. There can be up to $\binom{7}{4} = 7 * 6 * 5/(2 * 3) = 35$ fitness functions, and we sample 32 of them. Using these different fitness functions, the solution found by the GA passed the generalization test with 100% success, and the average evaluation time of one individual was not significantly different from the average evaluation time we obtained using a single initial position. Furthermore this multiple fitness principles enabled a broader search of the space, and maintained the genetic diversity. Each sub population has a different goal in mind, and is therefore led to explore different parts of the search space. Intuitively, the power of cross over as a creative process can be exploited by recombining solution of different "ethnical origin".

7 A pictorial demonstration

In this section we report simulation and results, using mainly pictures to describe the experiment. The set of alleles used was $\{S, P, T, A, G, H, R, C, D, I, U, L, W\}$. These alleles corresponds to program symbols of the cellular encoding: they have been explained in Section 2, except for alleles "I", "U", and "L". The program symbol "I" sets the input weight n to the value $+1$, where n is the argument. Alleles "U" and "L" differentiate a cell into an ANN's unit having sigmoids respectively s_{256} and s_{2048}, (s_α) is a family of piece-wise linear functions described in Section 4. Program symbols $\{C, D, I, U, L\}$ have arguments. The arguments of "C","D","I" is the number of a link. The argument of "U" or "L" multiplied by 256 gives the bias of the neuron.

Using these alleles, an initial random population is created, in which all the grammar trees have an equal and fixed number of genes. The arguments were set to random values between -9 and +9. The genetic algorithm is applied until a genetic code is found that develops a neural net meeting the termination criterion explained in preceding section, or the allocated time of two hours has elapsed.

The experiments are focussed on showing the interest of genome splicing. We did two kinds of experiments for a comparative study. In the first kind, the genome consists of a single grammar-tree and the crossover is done by exchanging sub trees. In the second kind the genome consists of three grammar trees numbered from 1 to 3. Cross over is done by exchanging sub trees between corresponding trees. Three subtrees are exchanged for one cross over. In this second experiment, we had to use another allele: the program symbol n, that has an argument. The argument of 'n' is a relative address for moving the reading head. It can be 1 or 2 for tree 1, it is always 1 for the tree 2, it does not exist in tree 3. By using a cellular code of three grammar trees instead of one we show that the GA can use the second or the third grammar tree to encode a sub neural network, and the first grammar tree to put together some copies of this sub neural net. We call that Automatic Definition of Sub Neural networks: ADSN. The GA ran on a 32 processor machine [Gruau 1993a]. In fact 32 GAs run in parallel, with complete independence and asynchronism. They randomly send individuals and look in their mailbox whether some of their neighbors have sent a "genetic parcel post" (an individual). On average, only one individual is sent each generations. We call individuals that are exchanged genetic parcel post, because they are rare and we hope that they bring a good surprise. To keep total independence between processors, we collect statistics separately for each processor and obtain 32 separate files, one for each processor. A practical interest of total independence is that the same GA runs on a Sun workstation, and the Ipsc860 with 32 processors. Hence, the parallel GA is easy to program, to debug and to maintain. Another interest of asynchronism is that communication is overlapped by computation, and does not cost any computer time.

A run of the GA also produces another 32 files recording the genetic code of the best-so-far individual found by the GA, for each processor. We can afterwards run a program that display the phenotype (ANN) of each of these champions, one after the other, by just clicking on the mouse button. We get the impression of seeing the neural network evolving in real time. Figures 5 and 6 report the sequence of the champions for a GA run without ADSN, and Figures 7 and 8 show the sequence of champions with ADSN. The processor which was chosen is the one which find the neural network that satisfied the termination criterion. The bottom right picture in Figure 5 shows the solution found without ADSN, the bottom right picture on Figure 7 shows the solution found with ADSN. In Figure 9 I show developmental steps of the neural net found without ADSN. In

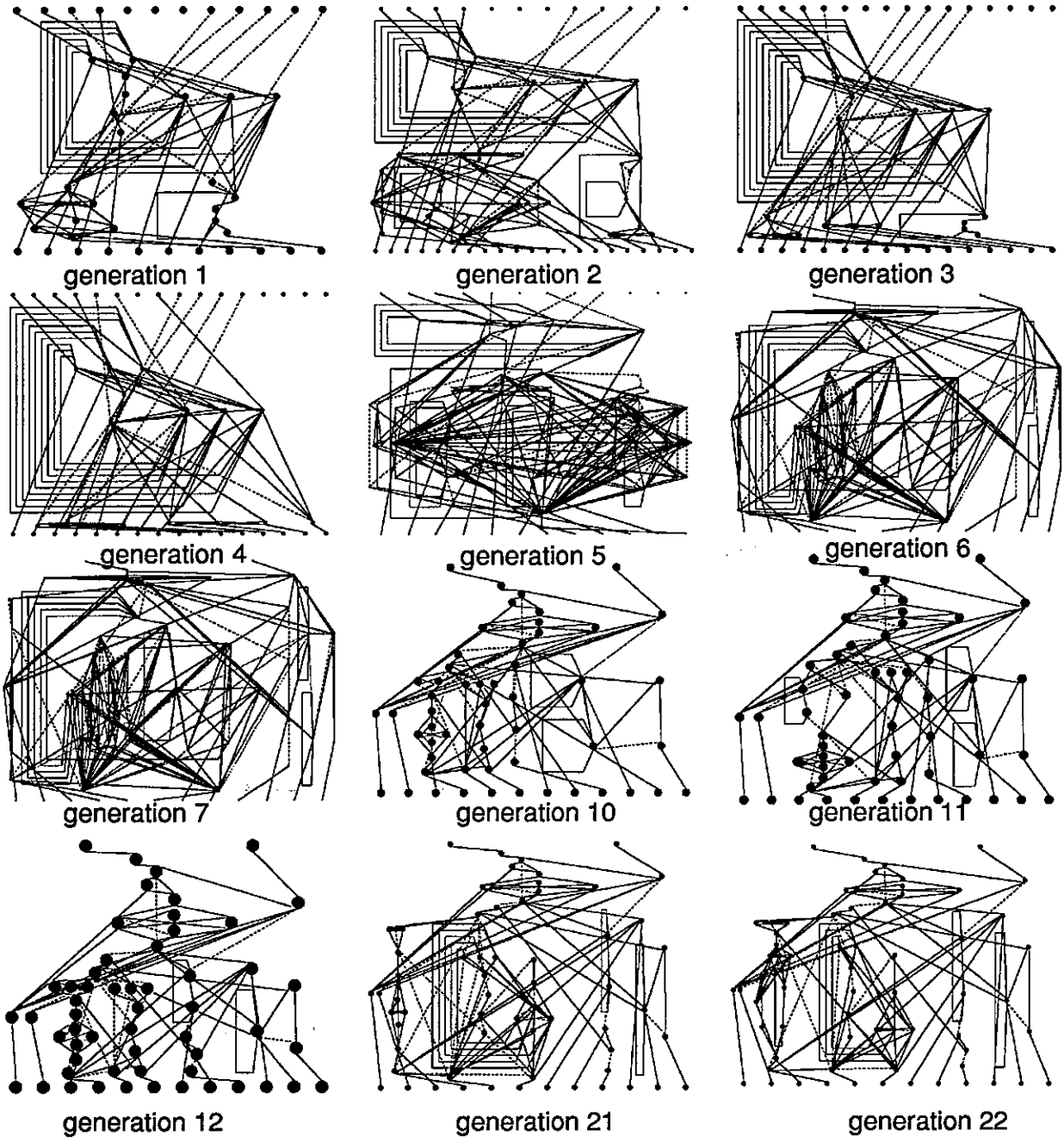

figure 5: evolution of the best individual during one run without ADSN

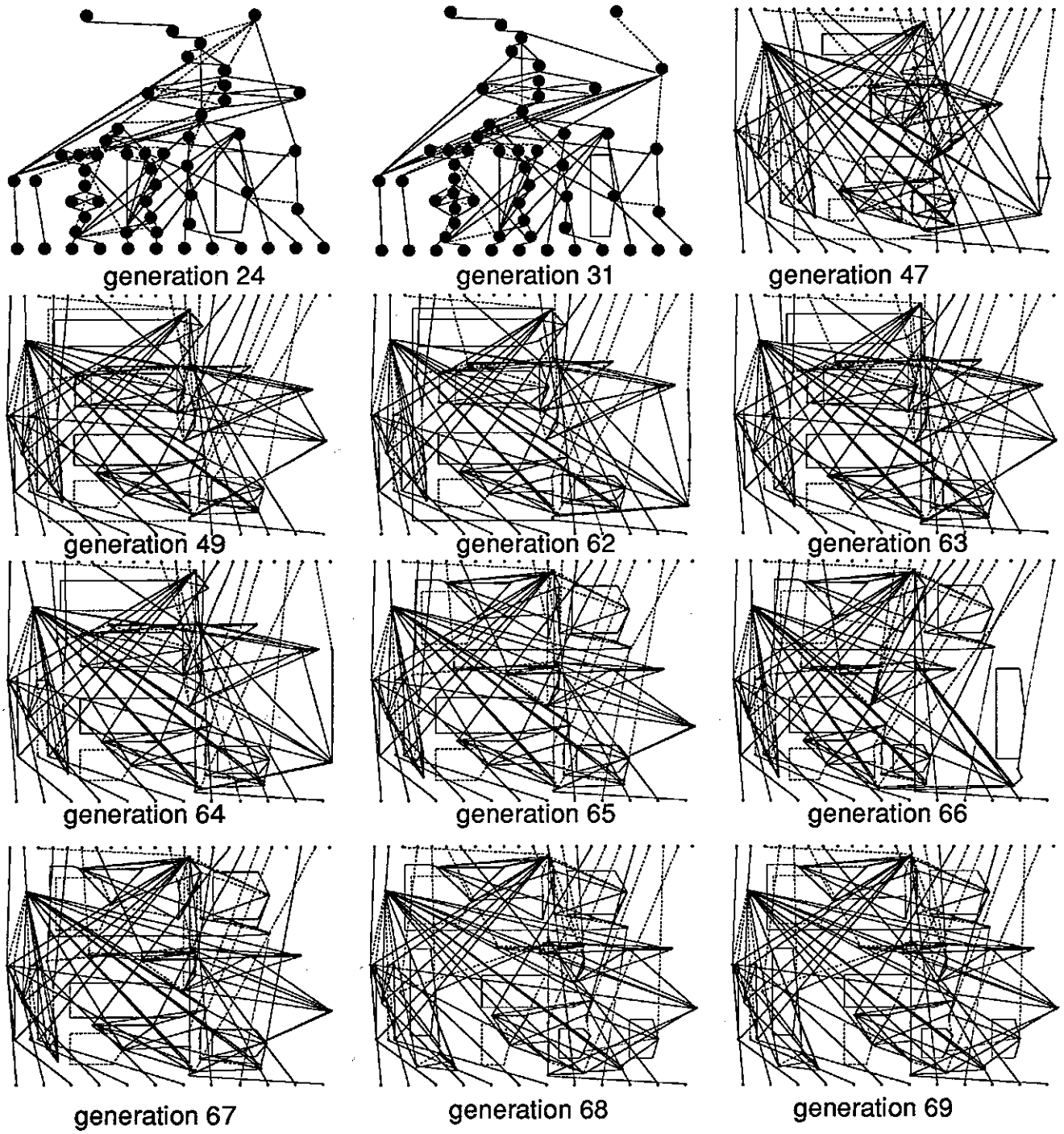

Figure 6: Evolution of the best individual during one run without ADSN (end)

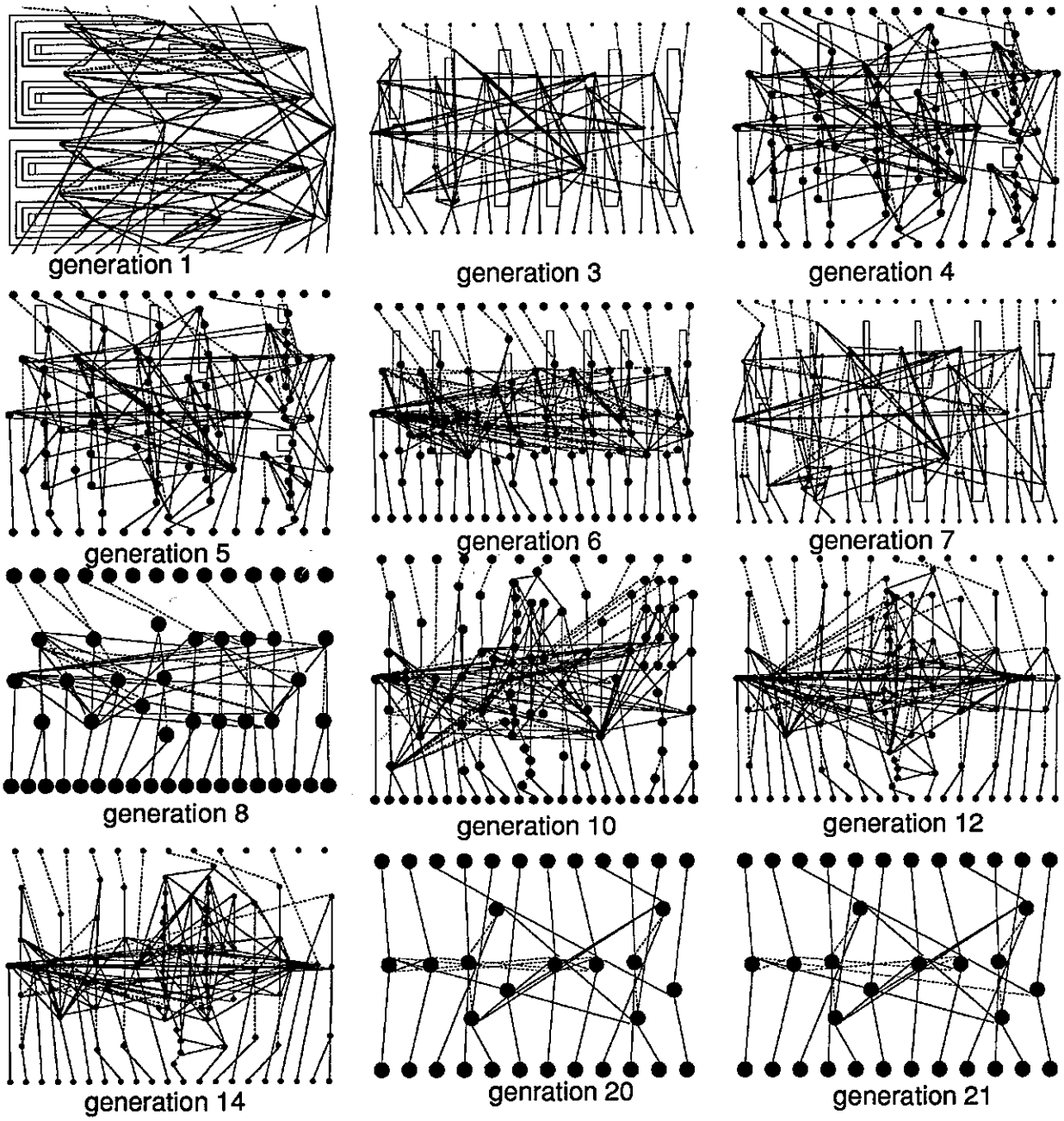

Figure 7: Evolution of the best individual during a run with ADSN (beginning)

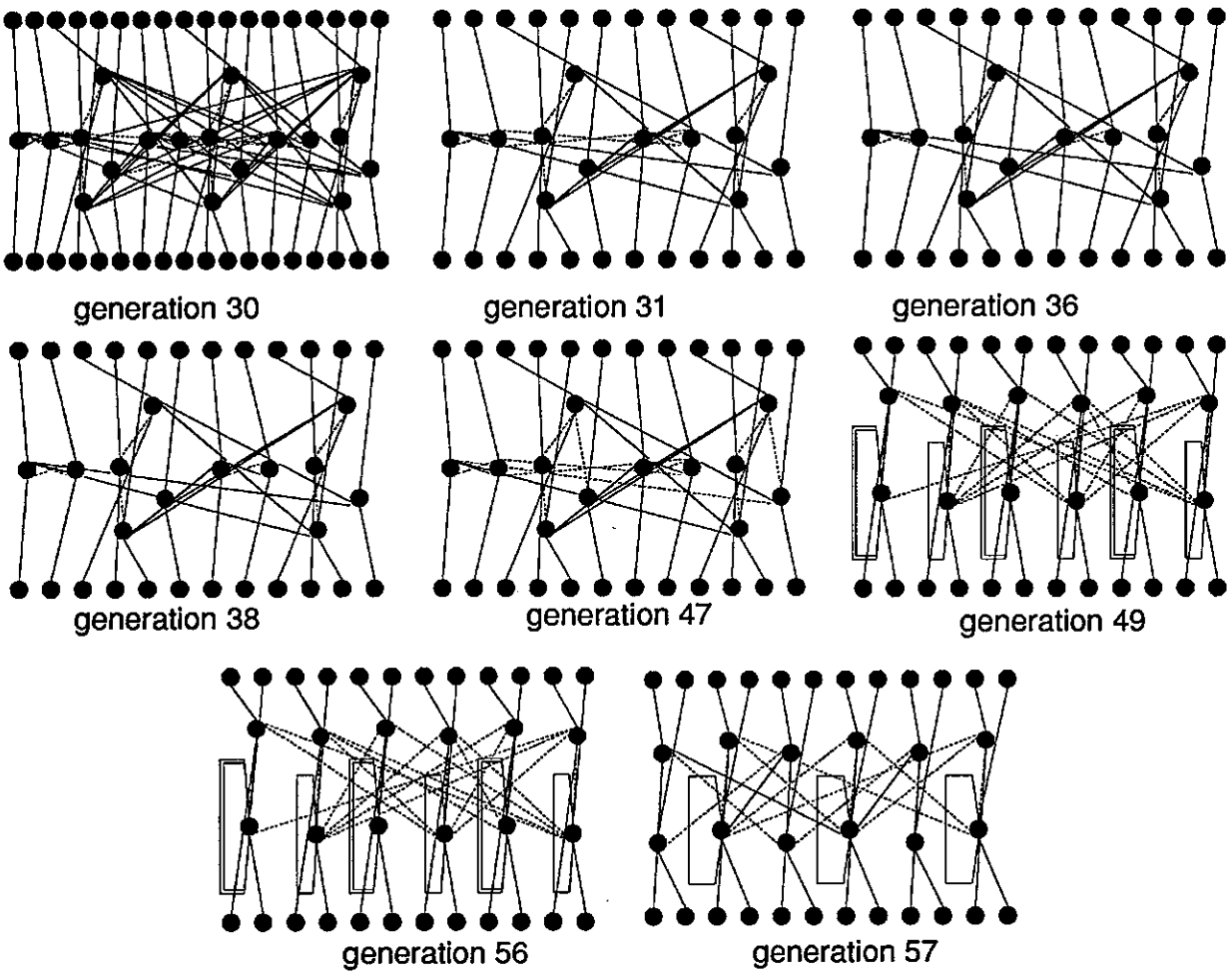

Figure 8:Evolution of the best individual during one run with ADSN (end)

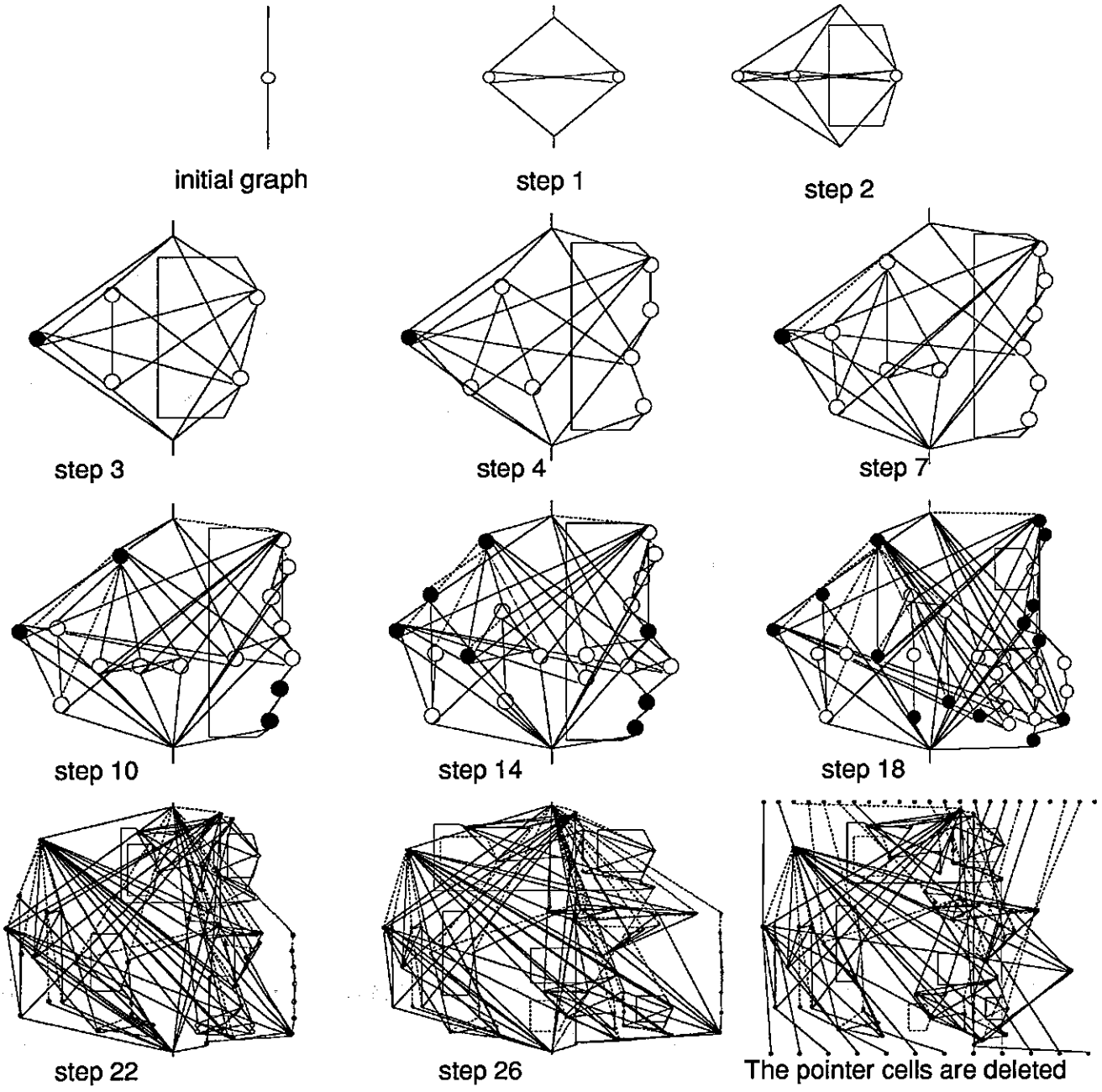

Figure 9: Development of a solution found by the genetic algorithm without ADSN

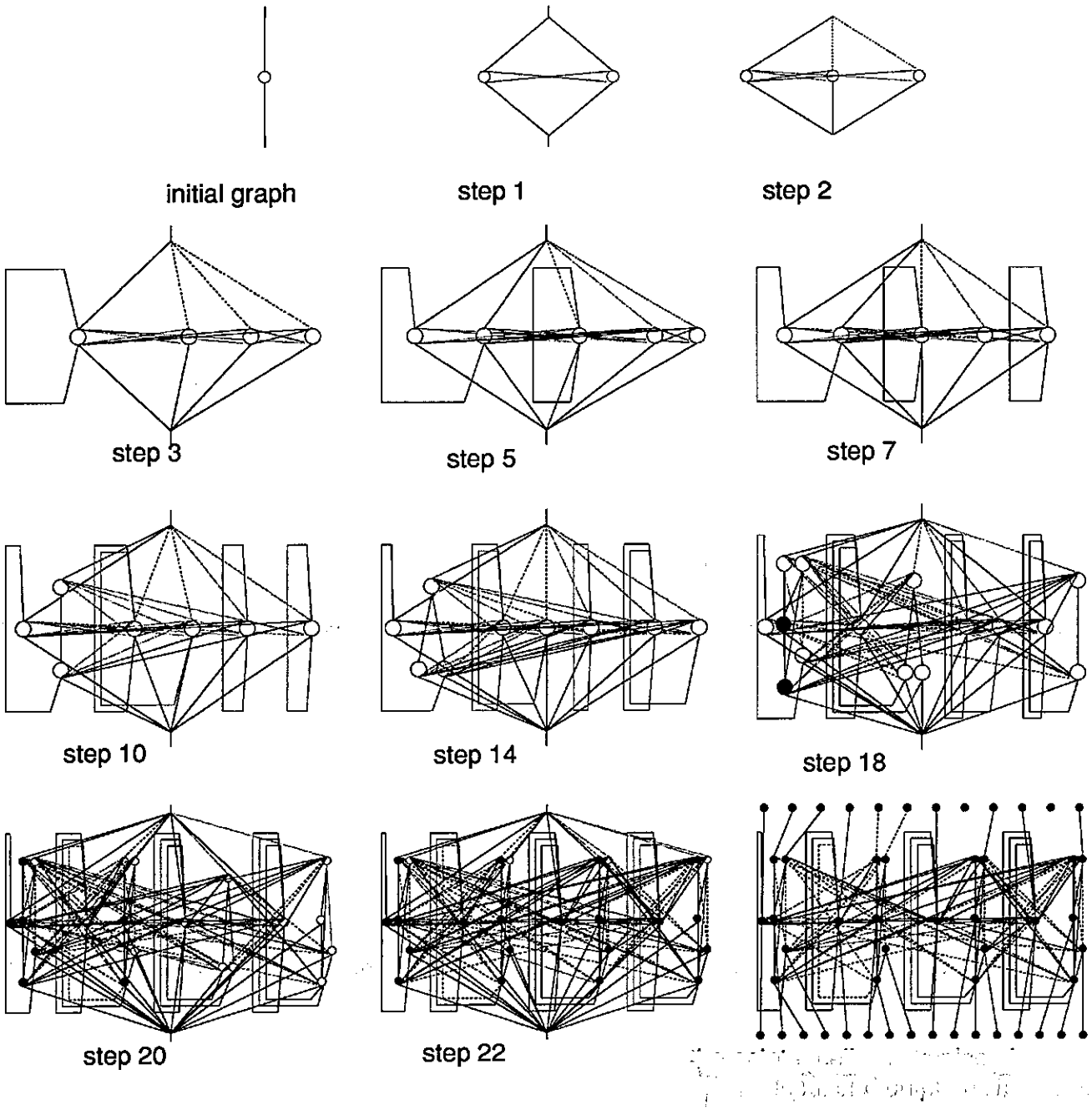

Figure 10: Development of a solution found by the genetic algorithm with ADSN

```
(a)
single tree:
A(A(U-3)(S(L3(M3(M9)))(P(T(L4(M1))(D-6(G(D2(P(L-8(F6(F1)))(S(L8(D-1(M3)))(I-5(M1
(F1))))))(L4(M3(M9))))))(A(U7(M2))(A(H(L-8(M1(L2(M3))))(S(W(M1))(U-5)))(C-3(T(L-
1(M1))(U-5)))))))))(R(T(S(L-2(M1(L4(L))))(T(D-8(W(F2)))(T(T(S(R(A(A(S(A(L-1)(L8))
(R(L3(M3))))(S(G(L-9(M1))(D7(M3(F1))))(L6)))(W(S(A(S(D-5(T(I9(U-1))(I-7(L(F1))))
)(S(L-6(M2))(U2()))))(R(L2(M1))))(T(L3(L-5))(L-7(M1)))))))(F1))(L-3(M2)))(L-3))))
(H(I8(P(S(H(P(L-2(M2(M1)))(R(U9(M1))))(S(U4)(U5(U-2(M1)))))(A(M1)(T(L5)(I-5(U-1)
))))(A(T(T(C(A(P(G(C(T(C-4(A(P(G(C6(L4(U-5(M1))))(W(U-8)))(I3(L-2)))(U-7)))(F1))
)(M1(L-6(L(M3)))))(U6))(U9)))(G(T(P(D1(L2))(M1))(T(D-2(L-6(M1)))(S(L8)(U-4(M1)))
))(R(T(M1)(C-3(L8))))))(W(W(T(A(A(S(A(L-3)(L-7))(R(L1(M3))))(S(G(L(M1))(D7(M3(F1
))))(L6)))(W(S(A(S(D(T(I-9(U-1))(I-7(L-3(F1)))))(S(L-6(M1))(U2(S(G(L(M1))(D7(M3(
F1))))(L6)))))(R(L2(M1))))(T(L3(L5))(L7(M1))))))(L-9(M1(M3)))))))(G(H(G(L2(M1(C9
(F1))))(D-2(L-1)))(D-1(D7(S(T(I(L6))(H(L8(L-2(M1(L-4(L-5)))))(L-5)))(U-9)))))(U2
)))))(G(U6)(M2)))))

(b)
tree 1: A(A(n2)(n2))(n2)
tree 2: not used
tree 3:
P(T(C-7(C-7(D5(M3(M2(C-7(C-7(D5(M3(M2(C-9(I2(U-5(C2(I8(U1(M2(F2(M2(I-1(F2(M4(U))
))))))))))))))))))))))))(D5(D5(M3(M2(I2(M2(I4(F4(F3)))))))))(T(C-7(C-7(C-7(D5(M3(M
2(I7(W(F2(M3(M2(I4(F2(F4)))))))))))))))(R(U6(M2))))
```

Figure 11: Comparative size of the genome found by the GA, (a): without Automatic Definition of Sub Network and (b): with ADSN (b). Without ADSN, the number of tree nodes inherited by learning is 57, and the total number of nodes is 260. With ADSN, the number of learned nodes is 19, and the total number of nodes is 65.

Figure 1 I show development steps of a neural net found with ADSN in another run. In Figure 11 I show the genotype with and without ADSN. In Figure 12 I show foot steps of the hand-coded solution, of the solution found with ADSN and of the solution found without ADSN. We ran two trials. with and without ADSN, on an Ipsc860 with 32 processors. The time limit was set to two hours; if after two hours no solutions are found, the GA is restarted. In all the runs, the robot first learns to generate oscillatory movements of the legs, and thereafter learns to coordinate the movements of the leg to avoid falling. With ADSN the GA found a solution in an average time of 1968 seconds, and an average number of 5152 evaluations. In both cases, the solution found by the GA was general: the ANN produces tripod gait over all the possible initial conditions of the legs, and the average distance made by the robot is the same (85 meters) as the hand coded neural network. Without ADSN, the GA could not find a solution during the first run, and did find a solution in the second, after one and a half hours. If we count the time for the failure which is the standard way to do comparison between two optimization schemes, the average time taken was

12752 seconds, and the number of evaluations: 18805. Therefore, the speed up brought by ADSN is 6.5 and the run without ADSN uses 3.65 times less individuals. Furthermore in Figure 10, we can see that the genetic code found without ADSN is more than four times bigger. Figures 5, 6, 7 and 8 show that the phenotypes are also bigger. The final neural net found without ADSN has 53 hidden units, and the one found with ADSN has 12 hidden units, more than four times less. Two trials are not sufficient to provide sufficient statistical information. However, the difference is so big between runs without ADSN and runs with ADSN that we think it does prove the efficiency of ADSN. The main reason why we did not perform more experiments was limited access to the parallel machine. The experiment with ADSN and without ADSN had exactly the same parameters, except for one difference: in the run without ADSN, the initial population consisted of individuals having 60 genes, and the upper bound on the size of the trees was 600 genes, and in the run with ADSN, the initial population consisted of individuals having three trees of 20 genes each, with upper bound 100 genes. In both case, the genetic material (number of genes) is the same in the initial population. We gave the possibility to grow two times as much genetic material to the run without ADSN. Since genetic code cannot be reused, we thought there needed to be more genetic material.

Figures 5,6,7 and 8 provide useful insights as to how the GA proceeds to builds the ANNs. There are some periods in which the general structure of the architecture remains the same and only a careful scrutiny reveals the few connections that have been changed. These changes are made by learning, or mutation and crossover towards the leaves of the trees. The impact of a change in the genotype is big if the change is made early in the development, near the root of the tree; it is small if it is made late in the development, near the leaves of the tree. At other periods, a deep change in the structure can be observed from one best individual to the next. This is due either to cross over near the root of the tree, or to the receipt of a good "genetic parcel post" from neighboring processors. Sometimes we can see two species of neural nets reappearing alternatively. These two species are competing for the podium of championship. With ADSN, the impact of crossover or mutation also depends on which of the tree is is applied. The cross over on the first tree changes the global structure of the ANN, how many copies of the sub network are produced and how they are combined. During the run with ADSN, at generation 30 (Figure 7), we see a marked transition toward a general structure which is composed of two oscillators. These structure will remain and be improved six times before another structure takes over at generation 49 (Figure 8). The new structure is made of three oscillators. After two improvements the three oscillators structure leads to a solution that satisfies the termination criteria. In the first improvement at generation 56, although the genotypes are different, the phenotype is the same, but the neuron were initialized with activities that lead to the tripod gait attractor more often. The second improvement at generation 57 is a pruning of misleading trees.

Whereas our hand-coded solution encompasses six oscillators, the GA could find a solution with only 3 oscillators. The other solution developed Figure 10 seems to have 4 oscillators, because the sub network is included four times. From Figure 11 we can compute the percentage of alleles that have been learned and inherited with the Lamarckian strategy. It is the proportion of alleles "M", "N" and "F" in the trees, since these particular alleles have been used for back-coding. 30% of the alleles of the solution found with ADSN come from learning. Without ADSN, the percentage drops to 22%. We tested the efficiency of random learning. We ran two trials of the same experiment with ADSN, but suppressing the learning step. We had two successes with an average time 2589, and an average number of 10724 evaluations . Hence the speed up brought by learning is 1.32. Figure 12 (a) and (b) show that the neural net found with ADSN has two different attractors, one

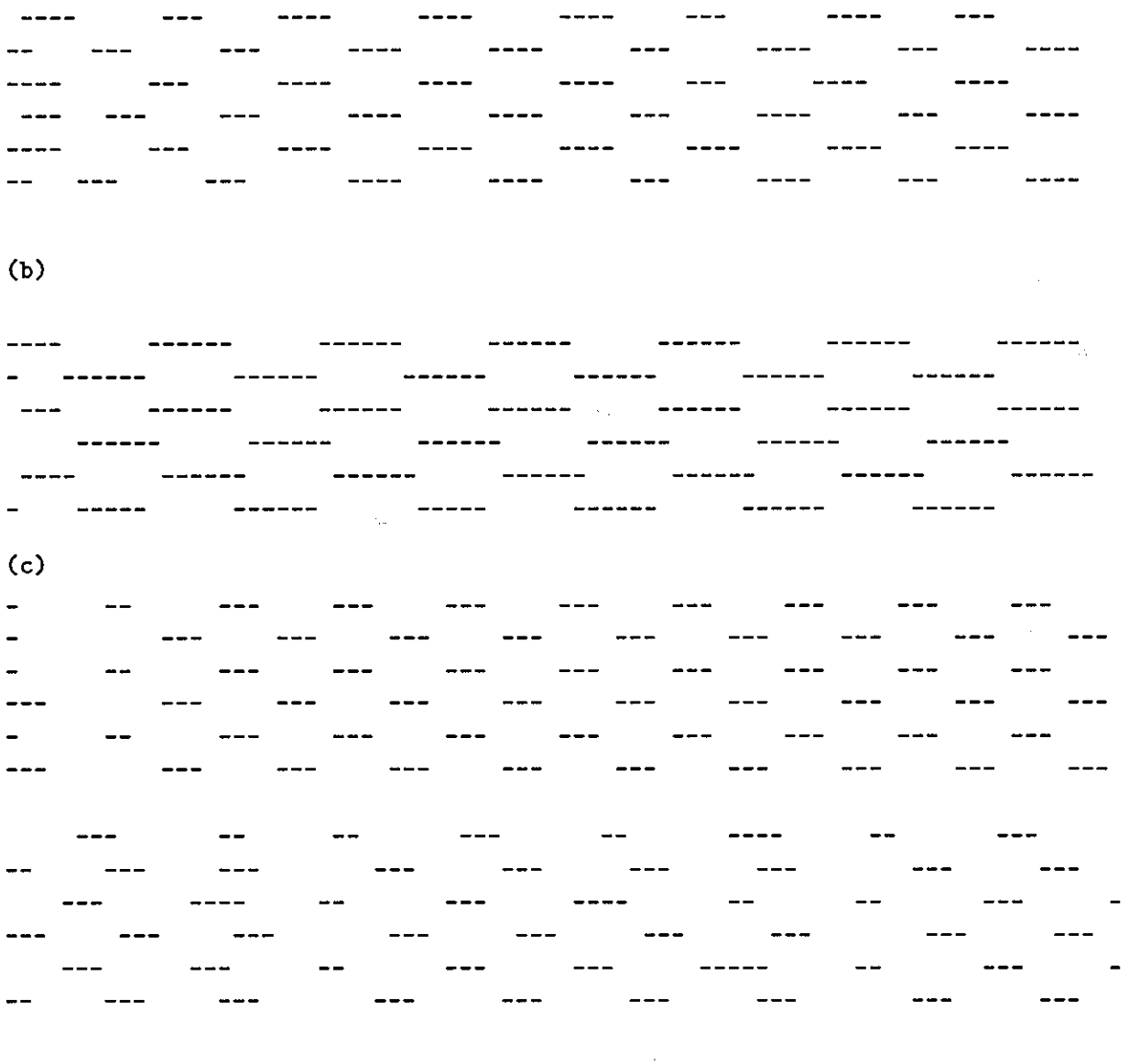

Figure 12: Illustration of foot steps. Foot steps are represented in the following way: whenever a leg is up, we plot a segment of line; otherwise nothing is plotted. The lines corresponding to the six legs are plotted one under the other, in the following order: left posterior, left middle, left anterior, right posterior, right middle, right anterior. Clearly the tripod gait is illustrated. (a) foot steps produced by the hand coded ANN (b) foot steps produced by the ANN found by the GA without ADSN. (c) two types of foot-steps produced by the ANN generated with ADSN.

in which the gait is completely regular, and one in which the gait is quasi regular, the same kind of quasi regular foot steps and regular foot steps are consistently observed over the 64 possible initial positions, with 0.5 probability each.

8 Conclusion

We have illustrated how artificial morphogenesis can be used to produce structured Artificial Neural Networks (ANN) that solve more complex problems, by exploiting regularities inherent in the problem. The genome is spliced into a list of trees which can refer to each other in a hierarchical way, and each subtree encodes a sub network, so that the total ANN is a hierarchy of many copies of these smaller sub networks. We did two kind of experiments. In the first kind, the structure of the chromosome is a single tree. In the second kind, the structure is a list of trees. The second experimental setting allows to find an ANN solution to a simplified six-legged locomotion problem 6.5 faster than the first one. Furthermore, the size of the genotype and the number of hidden units of the phenotypes (ANN) are both more than four times smaller in the second setting. Although the average was made on two trials, we believe the difference between the two trials is sufficiently high to show the superiority of the second setting.

We acknowledge that the idea and the "technology" of genome splicing comes from John Koza, back to a discussion at IJCNN Baltimore. Koza has now written a book [Genetic Programming II] where he demonstrates that genome slicing works, within the context of genetic programming, and we believe this book is going to make a little revolution. *Genetic Programming* has been demonstrated by Koza as a way of evolving computer programs with a GA. In the 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.

Genome splicing is called by Koza, "automatic function definition", because each tree encodes an Automatically Defined Function (ADF), and the solution is a hierarchy of functions that call each other many times. We call that "Automatic Definition of Sub Neural Networks", because with cellular encoding each tree encodes a sub neural network.

The idea of genome splicing is general, and we think that in lots of domains in the evolutionary algorithm community on could advantageously leave the "inert" fixed length bit string model, and splice the genome in order to give it life and dynamism. All what is needed to do genome splicing is a mechanism allowing the genome parts to reference each other. In [1994 b], we compared Lisp and Cellular encoding as two possible ways of splicing genomes. On one hand evolving ANNs instead of LISP expressions is computationally more expensive. It takes much longer to develop and evaluate a neural network than a LISP S-Expression. Koza uses a workstation, and I use a 32 processor parallel machine wich is probably 32 times more powerful, since one processor is roughly equivalent to Koza's machine. On the other hand, genome splicing, in cellular encoding, presents some advantages over genetic programming. First with GP one must specify for each ADF, the number of arguments that will be passed. We do not have to do that with cellular encoding. Second, Koza specifies a different set of alleles for the main program and the ADF; we do not need to do that either, because the set of alleles used in cellular encoding is homogeneous: cell division and modification of weights. Third because we generate ANNs, we can speed up the genetic search by combining the GA with a learning of the weights. In [Gruau 1994 a] we where able to speed up the genetic search by a factor of up to 13 using a supervised learning method. In this paper, we used a random learning that save 30% of the computer time. We are sure we can do better than a random

learning method. Fourth, genetic programming, in that it implies a symbolic search, suffers from the symbol grounding problem: how to map symbols to the real world. With cellular encoding, the search is conducted at a less abstract level.

Finally, the extra computer time needed for ANN's fitness evaluation may be the price to pay to tackle problems from a lower level of abstraction, climb up the hierarchy of complex organization, and come up with a more robust method, where one does not have to change the set of alleles for every new problem.

9 Acknowledgment

This work has been supported by the Centre d'Etudes Nucléaires de Grenoble, the European Community within the Working Group ASMICS and the Santa Fe Institue under the adaptive computation program. We thanks Oak Ridge National Laboratory for providing access to their 128 nodes Ipsc860. Bill Macready gave us the idea of having different fitness functions based on a sharing of the examples to learn, many thanks for that great idea. We thank Melanie Mitchell and Rajarshi Das for their precious comments and corrections and Darrell Whitley, Una-May O'Reilly, and Kenneth DeJong for fruitful discussions about this work.

References

[1] *International Conference on Graph Grammars.* Lecture note in computer Science 532, 1990.

[2] Randall Beer. *Intelligence as adaptive behaviour.* Academic Press, 1990.

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

[4] R. K. Belew, J. McInerney, and N. Schraudolf. Evolving networks, using the genetic algorithm with connectionist learning. Technical Report CSE-CS-90-174, UCSD, 1990.

[5] F. Gruau. Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In *Combination of Genetic Algorithms and Neural Networks*, 1992.

[6] F. Gruau. The mixed parallel genetic algorithm. In *Parallel Computing 93*, 1993 a.

[7] F. Gruau. Process of translation and conception of neural networks based on a logical description of the target problem. Patent EN 93 158 92, December 30, 1993 b.

[8] F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD Thesis, Ecole Normale Supérieure de Lyon, 1994. anonymous ftp: lip.ens-lyon.fr (140.77.1.11) directory pub/Rapports/PhD file PhD94-01-E.ps.Z (english) PhD94-01-F.ps.Z (french).

[9] F. Gruau and D. Whitley. Adding learning to the the cellular developmental process: a comparative study. *Evolutionnary Computation V1N3*, 1993.

[10] F. Gruau . Genetic Micro Programing of Neural networks *Advances in Genetic Programing,* Mit Press, editor: Kim Kinnear 1994 b.

[11] H. Kitano. Designing neural network using genetic algorithm with graph generation system. *Complex Systems*, 4:461–476, 1992.

[12] John R. Koza. *Genetic programming: A paradigm for genetically breeding computer population of computer programms to solve problems.* MIT press, 1992.

[13] John R. Koza. *Genetic programming II: Automatic discovery of reusable programs.* MIT press, 1994.

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

[15] D. Parisi and S. Nolfi. Growing neural networks. Technical report, PCIA 9115 University of Rome, 1991.