Cellular encoding for interactive evolutionary robotics

F. Gruau and K. Quatramaran

Computer Science/Department of Algorithmics and Architecture

CS-R9629 1996

Report CS-R9629
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Cellular Encoding for Interactive Evolutionary Robotics

Frédéric Gruau[1,2] and Kameel Quatramaran[1]

[1] *University of Sussex,*
School of Cognitive and Computing Sciences,
The Robotics Labs.
Falmer, Brighton, BN1 9QH UK
http://www.cogs.susx.ac.uk/

and

[2] *CWI*
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
gruau@cwi.nl, http://www.cwi.nl/ gruau/gruau/gruau.html

Abstract

This work reports experiments in interactive evolutionary robotics. The goal is to evolve an Artificial Neural Network (ANN) to control the locomotion of an 8-legged robot. The ANNs are encoded using a cellular developmental process called cellular encoding. In a previous work similar experiments have been carried on successfully on a simulated robot. They took however around 1,000,000 different ANN evaluations. In this work the fitness is determined on a real robot, and no more than a few hundreds evaluations can be performed. Various ideas were implemented so as to decrease the required number of evaluations from 1,000,000 to 200. First we used cell cloning and link typing. Second we did as many things as possible interactively: interactive problem decomposition, interactive syntactic constraints, interactive fitness. More precisely: 1- A modular design was chosen where a controller for an individual leg, with a precise neuronal interface was developed. 2- Syntactic constraints were used to promoting useful building blocs and impose an 8-fold symmetry. 3- We determine the fitness interactively by hand. We can reward features that would otherwise be very difficult to locate automatically. Interactive evolutionary robotics turns out to be quite successful, in the first bug-free run a global locomotion controller that is faster than a programmed controller could be evolved.

AMS Subject Classification (1991): 68T05
CR Subject Classification (1991): I.2.6
Keywords & Phrases: genetic algorithm, evolutionary robotics, cellular encoding, artificial neural networks, interactive design

1. Introduction

1.1 The motivation for Interactive Evolutionary Algorithm

In [3] Dave Cliff, Inman Harvey and Phil Husband from the university of Sussex lay down a chart for the development of cognitive architectures, or control systems, for situated autonomous agent. They claim that the design by hand of control systems capable of complex sensorimotor processing is likely to become prohibitively difficult as the complexity increases, and they advocate the use of Evolutionary Algorithm (EA) to evolve recurrent dynamic neural networks as a potentially efficient engineering method. Our goal is to try to present a concrete proof of this claim by showing an example of big (> 16 units) control system generated using EA. The difference between our work and what we call the "Sussex" approach is that we consider EAs as only one element of the ANN design process.

An engineering method is something which is used to help problem solving, that may be combined with any additional symbolic knowledge one can have about a given problem. We would never expect EAs to do everything from scratch. Our view is that EA should be used interactively in the process of ANN design, but not as a magic wand that will solve all the problems. In contrast with this point of view, Cliff Harvey and Husband seem to rely more on EAs. In [3] they use a direct coding of the ANN. They find ANN without particular regularities, although they acknowledge the fact that a coding which could generate repeated structure would be more appropriate. The advantage of the Sussex approach is that it is pure machine learning, without human intervention. In contrast, we use EA interactively in the ANN design. This is similar to supervised machine learning.

1.2 How do we supervise the evolutionary algorithm?

The key element that enables us to help the EAs with symbolic knowledge is the way we encode ANNs. What is coded is a developmental process: how a cell divides and divides again and generates a graph of interconnected cells that finally become an ANN. The development is coded on a tree. We help the EA by providing syntactic constraints, a "grammar" which restrict the number of possible trees to those having the right syntax. This is similar to program in C needing to satisfy the C-syntax. Syntactic constraints impose a prior probability on the distribution of ANN. One advantage of our approach is that we are able to study the structure of our ANN, identify some regularities, and help the emergence of them by choosing the appropriate syntactic constraints. We don't want to use neurons in a sort of densely connected neural soup, a sort of raw computational power which has to be shaped by evolutionary computation. Instead we want a sparsely connected structure, with hierarchy and symmetries, where it is possible to analyse what's going on just by looking at the architecture. We think one needs a lot of faith to believe that EAs can quickly generate complex highly structured ANNs, from scratch. Perhaps nature has proven it is possible, but it took a lot of time and a huge number of individuals. Our approach is to use symbolic knowledge whenever it is easy and simple. By symbolic we mean things which can be expressed by syntactic constraints which are formally BNF grammars. By easy we mean symmetries that anybody can perceive. We see symbols as a general format that can define the symmetries of the problem or decompose a problem into sub-problems, or else provide building blocks. The discovery of such things is time-expensive to automate with evolutionary computation, but easily perceived by the human eye. Any non-scientific person can point out the symmetries of the 8-legged robot, and thus build the "symmetry format". We view Evolutionary Computation of ANN as a "desing amplifier" that can "ground" this symmetry format on the real world. This is may be another way to address the well known symbol grounding problem.

1.3 The challenge of this work

It is possible to automate problem decomposition and symmetry detection. First of all we should say that we still do automatic problem decomposition in this work, the EA automatically decomposes the problem of generating 8 coupled oscillators, into the problem of generating a singleton, and putting together copies of it. However we give some information about the decomposition, since we provide the number 8. In [6] we show that the EA could alone decompose the 6-legged locomotion problem into the sub-problem of generating a sub-ANN for controlling one leg and put together six copies of the sub-ANN. There we did not give the number 6. We needed however a powerful IPSC860 32 processors parallel machine, and over 1,000,000 evaluation. We are now working with a real robot (see figure 0.1), and each fitness evaluation takes a few minutes, and is done by hand. The challenge of the paper was to solve the same problem with only a few hundreds of evaluations. At the outset, this did not seem promising. Four ideas made it possible:

- We enhanced the developmental process by adding cellular cloning and link typing,
- Using syntactic constraints, We forced three cloning divisions at the root of the cellular code, so as to be sure than an 8-fold symmetric network would develop.

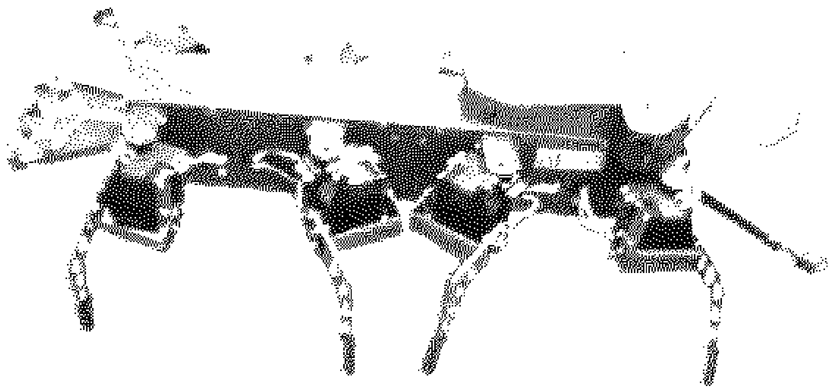

Figure 0.1: The OCT1 8-legged robot

- Each leg has two degrees of freedom, and two output units are needed to control them. We first evolved a leg controller to reduce those two outputs to a single output commanding the forward stroke and the return stroke. This way the problem is simplified to the task of generating 8 coupled oscillators.
- We determined the fitness by hand. We visually monitored the EA so as to reward any potentially interesting feature that would otherwise had been very difficult to detect automatically. We steer the EA starting from generating easy oscillatory behavior, and then evolving the leg coupling.

The paper presents what is cellular encoding, cell cloning and link typing, how we use syntactic constraints, experiments done with only one leg, and then with all the legs. Automatically generated drawing represent different ANN that were found at different stages of the evolution. We discuss the behavior of the robot, and try to explain it based on an analysis on the architecture, whenever possible. The description tries to render how it feels to breed robots.

2. Review of Cellular Encoding

Cellular encoding is a language for local graph transformations that controls the division of cells which grow into an Artificial Neural Network (ANN) [5]. Other kind of developmental process have been proposed in the literature, a good review can be found in [8]. Many schemes have been proposed with partly the goal of modeling biological reality. Cellular encoding has been created with the sole purpose of computer problem solving, and its efficiency has been shown on a range of different problem, a review can be found in [4]. We explain the basic version of Cellular Encoding in this section. A cell has an input site and an output site and can be linked to other cells with directed and ordered links. A cell or a link also possesses a list of internal registers that represent local memory. The registers are initialized with a default value, and are duplicated when a cell division occurs. The registers contain neuron attributes such as weights and the threshold value. The graph transformations can be classified into cell divisions and modifications of cell and link registers.

A cell division replaces one cell called the parent cell by two cells called child cells. A cell division must specify how the two child cells will be linked. For practical purposes, we give a name to each graph transformation; these names in turn are manipulated by the genetic algorithm. In the *sequential* division denoted `SEQ` 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 child cell. In the *parallel* division denoted `PAR` both child cells inherit both the input and output links from the parent cell. Hence, each link is duplicated. The child cells are not connected. In general, a particular cell division is specified by indicating for each child cell which link is inherited from the mother cell. The `FULL` division is the

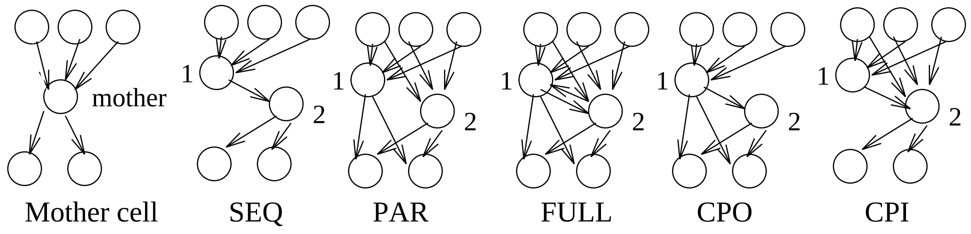

Figure 0.2: Illustration of main type of division: SEQ, PAR, FULL, CPO, CPI.

sequential and the parallel division combined. All the links are duplicated, and the two child cells are interconnected with two links, one for each directions. This division can generate completely connected sub-ANNs. The `CPO` division (CoPy Output) is a sequential division, plus the output links are duplicated in both child cells. Similarly, the `CPI` division (CoPy Input) is a sequential division, plus the input links are duplicated. Before describing the instructions used to modify cell registers it is useful to describe how an ANN unit performs a computation. The default value of the weights is 1, and the bias is 0. The default transfer function is the identity. Each neuron computes the weighted sum of its inputs, applies the transfer function and obtain s and updates the activity a using the equation $a = a + (s - a)/\tau$ where τ is the time constant of the neuron. See the figures 0.7 0.11 and 0.13 for examples of neural networks. The ANNs computation is performed with integers; the activity is coded using 12 bits so that 4096 corresponds to activity 1. The instruction `SBIAS` x sets the bias to $x/4096$. The instruction `DELTAT` sets the time constant of the neuron. `SACT` sets the initial activity of the neuron. The instruction `STEP` (resp LINEAR) sets the transfer function to the clipped linear function between -1 and $+1$ (resp to the identity function). The instruction PI sets the sigmoid to multiply all its input together. The `WEIGHT` instruction is used to modify link registers. It has k integer parameters, each one specifying a real number in floating point notation: the real is equal to the integer between -255 and 256 divided by 256. The parameters are used to set the k weights of the first input links. If a neuron happens to have more than k input links, the weights of the supernumerary input links will be set by default to the value 256 (i.e., $\frac{256}{256} = 1$).

The cellular code is a *grammar-tree* with nodes labeled by names of graph transformations. Each cell carries a duplicate copy of the grammar tree and has an internal register called a reading head that points to a particular position of the grammar tree. At each step of development, each cell executes the graph transformation pointed to by its reading head and then advances the reading head to the left or to the right subtree. After cells terminate development they lose their reading-heads and become neurons.

The order in which cells execute graph transformations is determined as follows: once a cell has executed its graph transformation, 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. The `WAIT` instruction makes a cell wait for a specified number of steps, and makes it possible to also encode a particular order of execution.

We also used the control program symbol `PROGN`. The program symbol `PROGN` has an arbitrary number of subtrees, and all the subtrees are executed one after the other, starting from the subtree number one.

Consider a control problem where the number of control variables is n and the number of sensors is p. We want to solve this control problem using an ANN with p input units and n output units. There are two possibilities to generate those i/o units. The first method is to impose the i/o units using appropriate syntactic constraints. At the beginning of the development the initial graph of cells consists of p input units connected to a reading cell which is connected to n output units. The input and output units do not read any code, they are fixed during all the development. In effective these cells are pointers or place-holders for the inputs and outputs. The initial reading cell reads at the root of the grammar tree. It will divide according to what it reads and generate all the cells that will eventually generate the final decoded ANN. The second method that we often prefer to use, is to have the EA find itself the right number of i/o units. The development starts with a single cell connected to the input pointer cell and the output pointer cell. At the end of the development, the input (resp. output) units are those which are connected to the input (resp. output) pointer cell. We let the evolutionary algorithm find the right number of input and output unit, by putting a term in the fitness to reward the network which have a correct number of i/o units. The problem with the first method is that we can easily generate an ANN where all the output units output the same signals, and all the inputs are just systematically summed in a weighted sum. The second method works usually better, because the EA is forced to generate a specific cellular code for each i/o unit, that will specify how it is to be connected to the rest of the ANN, and with which weights. To implement the second method we will use the instruction BLOC which blocs the development of a cell until all its input neurons are neurons, and the instruction TESTIO which compares the number of inputs to a specified integer value, and sets a flag accordingly. The flag is later used to compute the fitness.

Last, the instruction `CYC` is used to add a recurrent link to a unit, from the output site to the input site. That unit can then perform other divisions, duplicate the recurrent link, and generates recurrent connections everywhere.

3. Enhancement of Cellular Encoding

We had to enhance cellular encoding with cloning division, and the use of types. We also implemented another way to obtain recurrent links. All these new elements are reported in this section.

The cloning operation is really easy to implement, it is done by encapsulating a division instruction into a PROGN instruction. After the division, the two child cells only modify some registers and cut some links, then they simply go to execute the next instruction of the PROGN, and since they both execute the same instruction, it generates a highly symmetric ANN. Figure 0.3 represent a simple example of clone.

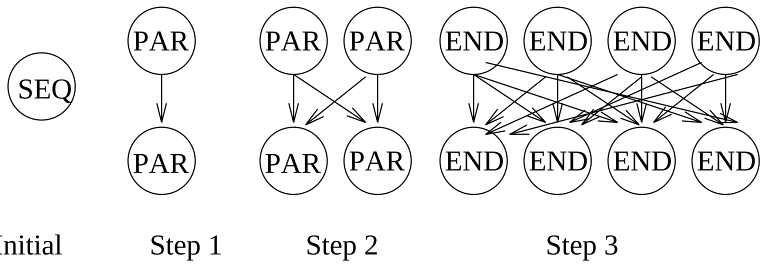

Figure 0.3: The cloning operation, the above ANN is developed from the code `PROGN(SEQ)(PAR)(PAR)(END)` which contains three clone division. The development takes three steps, one for each clone.

The instruction that we are now presenting have a version for the input unit which ends by the letter 'I' and one for the output units which ends by the letter 'O'. We are now going to use another link register called the `type` register, which will be initialized when a link is created between two child cells, and that is later used to select links for cutting, reversing, or setting the weight. We also introduce two sets of generic instruction one to select links, and another one to set link register.

The instructions beginning by 'C' and continuing by the name of a register r are used to select links. This instruction selects the links whose register is equal to the value of the argument. For example `CTYPEI(1)` selects all the input links for which the type register is equal to 1. There is another register called `NEW` which is a flag that is set each time a new link is created between two child cells, or a recurrent link is added. `CNEWO(1)` selects all the newly created links, going out of the output site.

The instructions beginning by 'S' and continuing by the name of a register are used to set the value of a register, in some previously selected links. For example the sequence `PROGN(CNEWI(1))(STYPEI(2))` sets the type register of newly created links from the input site, to the value 2. In this work, we only use this instruction for the weights, and for the type.

We also use the instruction `RESTRICTI` and `RESTRICTO` which has two arguments x and d_x. It is used to reduce a list of preselected links. Let say there are 10 input links whose type is 2. We can select the 5th and the 6th using the sequence `PROGN(CTYPEI (2))(RESTRICTI(5)(2))`.

The type register together with the select and the set instructions can be used to encode the connections from the neurons to the input and output pointer cell. Those connection are crucial, since they determine which are the input and the output units. Using type registers, we can let each neuron individually encode whether it is or it is not an input or an output unit. We assign two different types, say 0 and 1, to the link that links the ancestor cell to respectively the input pointer cell and the output pointer cell. We ensure that each time a cell divide, the links get duplicated, so that at the near end of the development, all the cells are connected to both the input and the output pointer cell. But then we ensure that each cell can potentially cut the links whose type are 0 or 1. In other words, if a cell wants to be an input unit, it just does nothing, but if it does not want, then it has to cut its input link of type 0.

Last, we use another instruction to add recurrent links. The instructions `REVERSEI` and `REVERSEO` duplicates a previously selected link from cell a to cell b, and changes the direction of the duplicate, make it go from cell b to cell a.

4. Syntactic Constraints

We used a BNF grammar as a general technique to specify both a subset of syntactically correct grammar-trees and the underlying data structure. The default data structure is a tree. When the data structure is not a tree, it can be `list`, `set` or `integer`. By using syntactic constraints on the trees produced by the BNF grammar, a recursive nonterminal of the type `tree` can be associated with a range that specifies a lower and upper bound on the number of recursive rewritings. In our experiments, this is used to set a lower bound m and an upper bound M on the number of neurons in the final neural network architecture. For the `list` and `set` data structure we set a range for the number of elements in these structures. For the `integer` data structure we set a lower bound and an upper bound of a random integer value. The `list` and `set` data structures are described by a set of subtrees called the "`elements`." The `list` data structure is used to store a vector of subtrees. Each of the subtrees is derived using one of the elements. Two subtrees may be derived using the same element. The `set` data structure is like the `list` data structure, except that each of the subtrees must be derived using a different element. So for example, the rule

$$< A >::= (list[2..2]of(0)(1))$$

generates the trees ((0)(0)), ((0)(1)), ((1)(0)), ((1)(1). The rule

$$< A >::= (set[2..2]of(0)(1))$$

generates only the trees ((0)(1)) and ((1)(0)).

Figure 4 shows a simple example of syntactic constraints used to restrict the space of possible solutions. The nonterminal <nn> is recursive. It can be rewritten recursively between 0 and 8 times. Each time is it rewritten recursively, it generate a division and adds a new ANN unit. Thus the final number of ANN units will be between 1 and 9. Note that in this particular case, the size of the ANN is proportional to the size of the genome, therefore constraints of the grammar in fig. 4 which controls the size of genome result directly in constraints on ANN growth which controls the ANN size.

The nonterminal <attribute> is used to implement a subset of four possible specializations of the ANN units. The first 8 weights can be set to values between -1 and $+1$. The time constant can

```
<nn>[0..8];
<axiom> ::=  <nn>

<nn>  ::= ( PAR(<nn>)(<nn>) )
            | (  CPO(<nn>)(<nn>) )
            | ( SEQ (<nn>)(<nn>) )
            | (  <attribute> )

<attribute> ::=
          (PROGN : set[0..4] of
              (WEIGHT: list[8..8] of
                   (integer[-255..+255]))
              (DELTAT(integer[1..+40]))
              (SBIAS(integer[-4096..+4096]))
              (STEP) )
```

Figure 0.4: Tutorial example of syntactic constraints

be set to a value that ranges from 1 to 40, and the bias is set to a value between -1 and $+1$. The transfer function can be set to the STEP function instead of the default transfer function. Since the lower bound on the `set` range is 0, there can be 0 specializations generated, in which case the ANN unit will compute the sum of its inputs and apply the identity function. Because the upper bound on the `set` is 4, all the 4 specializations can be generated. In this case, the neuron will make a weighted sum, subtract the bias, apply the clipped linear function. If the lower and the upper bound had been both 1, then exactly one and only one of the feature would be operator. This can be used to select an instruction with a given probability. For example, the sequence `PROGN: set [1..1] of (WAIT) (WAIT) (WAIT) (CYC)` generates a recurrent link with a probability of 0.25.

Crossover. Crossover must be implemented such that two cellular codes that are syntactically correct produce an offspring that is also syntactically correct (i.e. that can be parsed by the BNF grammar). Each terminal of a grammar tree has a primary type. The primary label of a terminal is the name of the nonterminal that generated it. Crossover with another tree may occur only if the two root symbols of the subtrees being exchanged have the same primary label. This simple mechanism ensures the closure of the crossover operator with respect to the syntactic constraints.

Crossover between two `trees` is the classic crossover used in Genetic Programming as advocated by Koza [9], where two subtrees are exchanged. Crossover between two `integers` is disabled. Crossover between two `lists`, or two `sets` is implemented like crossover between bit strings, since the underlying arrangement of all these data structures is a string.

Mutation. To mutate one node of a tree labeled by a terminal t, we replace the subtree beginning at this node by a single node labeled with the nonterminal parent of t. Then we rewrite the tree using the BNF grammar. To mutate a `list, set` or `array` data structure, we randomly add or suppress an element. To mutate an integer, we add a random value uniformly distributed between $\pm max(2,(M-n)/8)$. M and m are the upper and lower bounds of the specified integer range.

Each time an offspring is created, all the nodes are mutated with a small probability. For `tree`, `list` and `set` nodes the mutation rate is 0.05, while for the `integer` node it is 0.5. Those probability may be reset at run time of the EA.

5. The leg controller

5.1 The challenge

The leg does a `power stroke` when it pushes on the ground to pull the body forward, and the `return stroke` when it lifts the leg and takes it forward. The challenge in this first experiment was to build a good leg controller, one that does not drag the leg on the return stroke, and that starts to push on the ground right at the beginning of the power stroke. The ANN had one single input. The input of 4096 on the input unit must trigger the power stroke, and the input of 0 must trigger the return stroke That implies the right scheduling of four different actions: when exactly the neuron responsible for the lifting and the swinging lift up and down, swing forward and backward.

5.2 General setting

The command of return stroke or power stroke was hand generated during the genetic run, so as to be able to reproduce the movement whenever and as many times as desired. The EA used 20 individuals, the fitness was given according to a set of features: the highest and the lowest leg position had to be correct, the movement of the leg must start exactly when the signal is received, there must not be dragging of the leg on the return stroke, so the leg must first be lifted and then brought forward. Second the leg must rest on the floor at once on the power stroke, therefore the leg must be first put on the floor and then moved backward. Each of these features determined a range of fitness, the ranges where chosen in such a way that for all the features that were checked, the intersection of the ranges was not empty. The fitness was then adjusted according to a subjective judgement. The ranges evolved during the run, so as to fit the evolution. The EA was run around 30 generations. Fitness evaluation is much quicker than with the complete locomotion controller, because we have to watch the leg moving for only a very short period of time to be able to assess the performance, That is how we did up to six hundred evaluations.

5.3 Syntactic constraints used for the leg controller

We now comment on the syntactic constraints used in this run, which are described in figure 5.2. We did not used link typing or clone instructions for the leg controller, the ideas to use them came to us when we began to tackle the problem of evolving the whole locomotion controller. The non terminal <nn> generates one neuron, each time it is rewritten, since it can be rewritten between 6 and 20 times, the number of neurons will be between 7 and 21. The division instructions are classic, except for the SHARI1 where the input links are shared between the two child cells, the first child gets the first input, and the second child gets the other inputs. The ANN begins by generating 14 fake output units using parallel division (one clone and 7 normal). Those units reproduce the input signal. In this way, we can compare the movement on the leg whose controller is evolved, with the raw signal that moves the 7 other legs. The non-terminal <nn> is rewritten between 6 and 20 times, and finally the neuron specializes either as a temporal unit (non-terminal <t-unit>) or as a spatial unit (non-terminal <s-unit>.) The temporal units have a threshold sigmoid and a time constant that is genetically determined. They are used to introduce a delay in a signal, and the spatial units have a linear sigmoid, and a bias that is genetically determined. They are used to translate and multiply a signal by genetically specified constants. Those two types of units are the building blocks needed to generate a fixed length sequence of signals of different intensities and duration. The duration is controlled by the temporal units, and the intensity by the spatial units.

5.4 Explanation of the solutions

Figure 0.7 presents the leg controller found by the Evolutionary Algorithm, together with four different settings of the activities inside the ANN, and the corresponding leg positions. Figure 0.6 shows its genetic code after some hand-made obvious simplifications, such has merging a tree of PROGNs.. Neuron e controls the lift, neuron f controls the swing and neuron a is the input neuron. While the ANN is completely feed-forward, yet it can generate a short sequence of different leg commands. Because we use neurons with time constants, different events can happen at different time steps. In Step 1, the input to the ANN is 0, and the leg is forward down, the input is then changed to 4096 to initiate the power stroke. Step 2, we are in the middle of the power stroke, neuron e receives an even more negative input, this has no effect, since it was already over negative, the leg just stays on the ground. On the other hand, neuron f is brought to negative value, so the leg goes backward relative

```
<nn>[6..20];
begin
<axiom> ::= (LABEL
  (SEQ (WAIT) (PAR
       (<nn>)
      (PROGN
         (PAR)
         (PAR(PAR(PAR(WAIT)(WAIT))(WAIT))(PAR(WAIT)(PAR(WAIT)(PAR(WAIT)(WAIT)))) )  ) ) ) )

<nn> ::=   (SEQ(<nn>)(<nn>))
         | (PAR(<nn>)(<nn>))
         | (SHARI1(<nn>)(<nn>))
         | (CPI(<nn>)(<nn>))
         | (CPO(<nn>)(<nn>))
         | (FULL(<nn>)(<nn>))
         | (<tunit>)
 | (<sunit>)

<tunit> ::= (PROGN  (STEP)
              (PROGN : set[1..3] of
            ( DELTAT (integer[1..40]))
    ( WEIGHT: list[8..8] of ( integer[-256..+256]) )
    ( SBIAS  (integer[-4096..+4096]))  )   )


<sunit> ::= (PROGN
                (LINEAR)
                (PROGN : set[2..2] of
    ( WEIGHT: list[8..8] of ( integer[-1024..+1024]) )
    ( SBIAS  (integer[-4096..+4096]))  )   )
```

Figure 0.5: Syntactic constraints used for the leg controller

```
 SEQ (PROGN(STEP) (DELTAT(1))) (CPI  (SEQ (PROGN(LINEAR)(WEIGHT(-693 )
(-1024 )(360 )(-252 )(-300 )(-984 )(-849 )(610 ))(SBIAS(3497 )))
(CPO (PROGN(STEP)(SACT(3458 )))  (PROGN(STEP)(DELTAT(39 ))) ) ) (PAR
(PROGN(LINEAR)(WEIGHT(-693 )(-1024 )(360 )(-252 )(-300 )(-984 )(-706 )
(796 ))(SBIAS(1864 ))) (PROGN(LINEAR)(WEIGHT(-914 )(40 )(736 )(-622 )
(-1024 )(-984 )(-706 )(610 ))(SBIAS(-2321 ))) )  )
```

Figure 0.6: The genetic code of the Champion leg controller

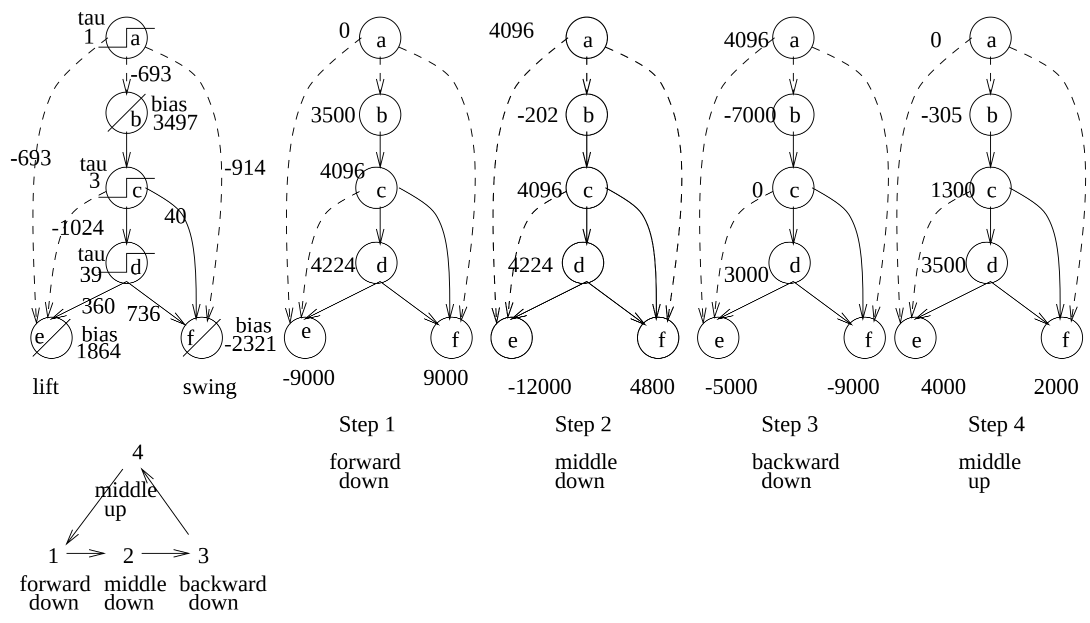

Figure 0.7: The leg controller, and four different settings of the activities inside the ANN, together with the corresponding leg positions. The default values are not represented. The default weight is 256, the default bias is 0, and the default time constant is 3. The diagonal line means the linear sigmoid, the stair case means the step function sigmoid.

to the body, and since the leg is on the ground, the body moves forward. Step 3, the power stroke is finished, we now initiate the return stroke the input is set to 0. Step 4, we are in the middle of the return stroke, neuron a is 0, but neuron c had not yet time to increase its activities, because of the time constants. Therefore, neuron e is positive, and the leg is up in the middle. When the return stroke terminates, we are back to step 1. Neuron c being positive force the neuron e to become again negative, and the leg is back on the ground. Initially we wanted to terminate the return stroke with the leg up, and to bring it down on the power stroke, the controller evolved in another way, and we thought it would be acceptable. We realized later, that it is actually much better this way, because the robot has always his leg on the ground, except when it is in the middle of a return stroke. So it does not loose balance often. There is another unexpected interesting feature of this controller. We put a security on the robot driver, so that at each time step the leg cannot move more than a small amount, this was to avoid warming of the servo motors. The evolutionary algorithm used this feature. It generates extreme binary leg position. Nevertheless, the leg moves continuously, because it cannot move more that the predetermined upper bound. Believe it or not, that was totally unexpected.

6. The locomotion controller

6.1 The Challenge

We have previously evolved ANN for a simulated 6-legged robot, see [6]. We had a powerful parallel machine, an IPSC860 with up to 32 processors. We needed 1,000,000 of ANNs to find the an ANN solution to the problem. In this study, one ANN takes a few minutes to assess the fitness, because the fitness is manually given, and it takes some time to see how interesting the controller is. One time we even spent an hour trying to figure out whether the robot was turning or going straight, because of the noise that was not clear. The challenge of this study was to help the EA so as to be able to more efficiently search the genetic space and solve the problem with only a few hundreds of evaluations instead of one million.

```
<axiom>::=(LABEL(SEQ
 (SEQ
  (PAR(<command>)(<command>))
  (PROGN
     (WAIT(4))  (CTYPEI(-1)) (RESTRICTI(0)(1)) (STYPEI(0)) (CTYPEI(-1))
     (STYPEI(1)) (CTYPEO(-1))  (STYPEO(0))
     (<evolved>)   )   )
  (PROGN
     (BLOC)
     (TESTIO8)
     (SHARI (JMP12)  (SHARI (JMP12)  (SHARI (JMP12)   (SHARI (JMP12)  (PROGN (SWITCH) (SHARI (JMP12)
          (PROGN (SWITCH) (SHARI (JMP12)   (PROGN (SWITCH) (SHARI (JMP12) (JMP12)
           (1) ))  (1) ))   (1) ))  (1) )   (1) )   (1) )   (1) )  ) ))
```

Figure 0.8: Syntactic constraint specifying the general structure

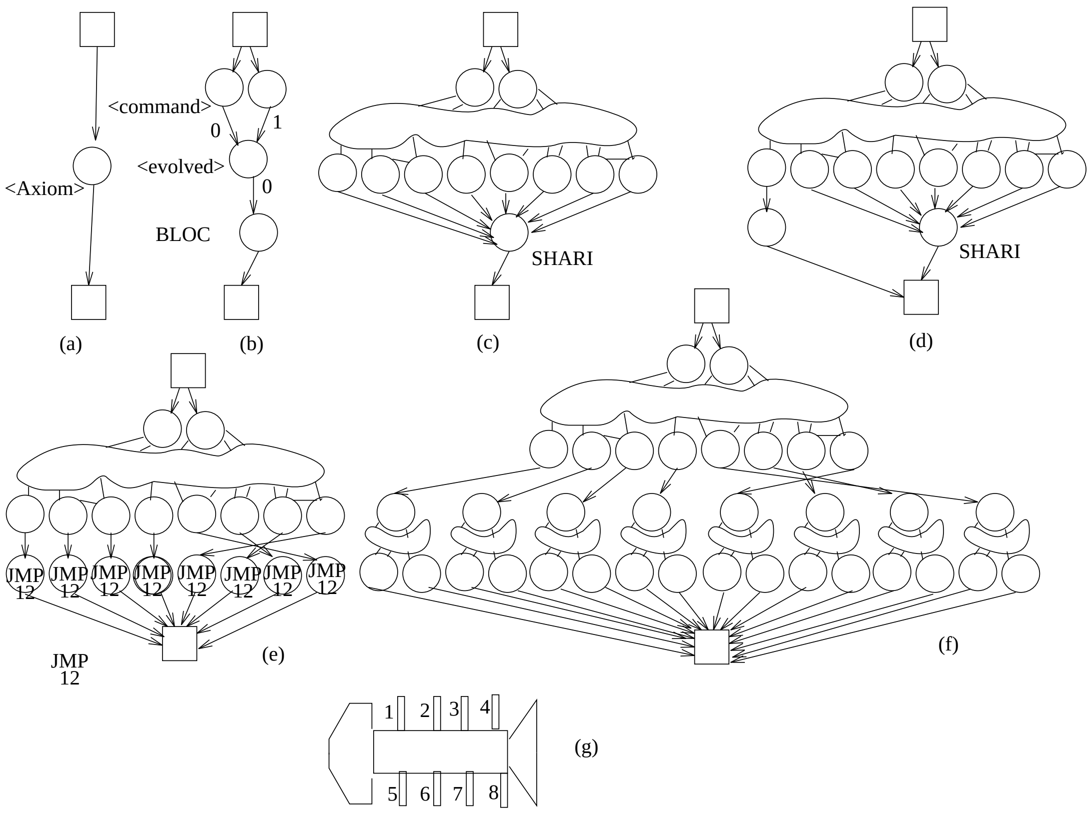

Figure 0.9: What the Syntactic constraint specifying the general structure do: The initial situation is (a), with the two pointer cells represented by a square, and the ancestor cell by a circle. (b) The general structure is specified after the three first divisions, two input units will execute a genetic tree generated by the non-terminal <command>, which is not used anyway. The central cell will develop the core of the controller, the bottom cell waits for the core to develop, then in un-blocs, check that it has exactly 8 neighbor, (it is the case here, but it could not, and then makes 7 SHARI division in (d) and (e), The last four divisions are interleaved with SWITCH operator, so as to set the leg numbers as is indicated in (g). Finally the 8 child cells make a JMP 12, where 12 is just the number of the tree where the cellular code that encodes a leg controller has been stored.

6.2 General setting

There are some settings which were constant over the successful run 2, and run 4, and we report them here. The way we give the fitness was highly subjective, and changed during the run depending on how we felt the population had converged or not. We realized that the role of the fitness is not only to reward good individuals, but also to control genetic diversity. Since there is a bit of noise when two individuals are compared, the selective pressure can be controlled by the fitness. The rewarding must be done very cautiously, otherwise newly fit individual will quickly dominate the population. We are quite happy to see a new good performing phenotype, and are inclined to give it a good rank, to be sure that it is kept for a while. However, if we see that same guy reappear again and again, we may kill it to avoid dominence. We followed some general guidelines. The fitness was a sum of three terms varying between 0.01 and 0.03: The first term rewards oscillations, and discourages too slow or too quick oscillations. The second term rewards the number of legs which oscillate, the third term rewards the correct phase between the different leg oscillators, and the correct coupling. We were very afraid to give big fitnesses, and wanted to put fitnesses difference in the range of the noise that exists when a Boltzmann tournament takes place. So our fitness seldom went above 0.1 .

We started with a population of 32 individuals so as to be able to sample building blocks, and reduce it to 16 after 5 generations. At the same time as we reduced the population, we lowered all the mutation rates (set list and tree) to 0.01 except the integer mutation rate which was increased to 0.4, the idea was to assume that the EA had gotten the right architecture, and to concentrate the genetic search on the weights. The weights were mutated almost one time out of two. The selective pressure was also augmented: the number of individual participating in Boltzmann tournament was increased from 3 to 5. Those tournaments are used to delete individuals or to select mates. We no longer wanted genetic diversity, but rather genetic convergence of the architecture, similar to tuning an already working solution. We got the idea to use genetic convergence from the SAGA paradigm of Inman Harvey [7]. It is possible to give a fitness -1, the effect is that the individual is immediately thrown to the garbage, and not counted. We threw away the motionless individuals, and those who had not the right number of input/output units. As a results, to generate the first 32 individual in the initial population, we go over one hundred evalutations.

6.3 Syntactic constraints used in all the runs

The first part of the syntax remains the same in the different run. It is represented in figure 0.8. It just specifies some fixed cellular code that is always to be part of any individuals. This codes a general structure which is developed in figure 0.9. We now details the execution of the code. First it sets the type of the links to the output pointer cell to 0 and develops an ANN with two input units typed 0 and 1. It then generates a cell which will execute the beginning of the evolved code, (<evolved>), and another cell blocked by the `BLOC` instruction. The input cells are not used. The blocked cell waits that all its neighbors are neurons, then it unblocks and executes the `TESTIO` instruction which has the effect to check the number of inputs, here it test whether it is equal to 8, and set a flag accordingly. This flag will be used in the fitness evaluation, to throw away those ANNs which does not have exactly 8 outputs units. The unblocked cell then goes on to execute the cellular code that develop the previously evolved leg controller at the right place. For this it used an ad-hoc division called 'SHARI' and also the 'SWITCH' operator which is used to assign number to the output units that match a logical numbering of the legs. This order is specified in figure 0.10 (g).

7. Log of the experimental runs

We did only five run, and we are proud of it. That is a proof that the method works well, it doesn't need weeks of parameter tuning. That's useful because one run is about two days of work. So we report the five runs, even if only one of them was really success-full, and the other were merely used to debug our syntactic constraints.

7.1 Analysis of run 0 and run 1

The first two runs were done with only seven legs, because a plastic gear on the eighth leg had burned, the robot had stubbornly tried to run into an obstacle for 30 seconds. As a result, ANNs were evolved that could made the robot walk with only seven legs. Run 0 brought an approximative solution to the problem. But after hours of breeding we input accidentally a fitness of 23, which had the effect to stop the EA, the success predicate being that the fitness is greater than 1. Run 1 also gave a not so bad quadripod, as far as we can remember, but we realized there was a bug in the way we used the link types which were not used at all. Instead of selecting a link and then setting the type we were first setting and then selecting, which amounts to a null operation. We had simply exchanged the two alleles. When we found out the bug, we were really surprised that we got a solution without types, but we think that's an interesting feature of EAs, even if your program is bugged, the EA will take care of it!.

7.2 Syntactic Constraints used in the second run

The syntactic constraints used for the second run are reported in 0.10. The core of the controller is developed by the cell executing the non-terminal <evolved>. This cell makes clones, and then generates a sub-ANN. The non-terminal <clone> is rewritten exactly 3 times, and generates exactly 3 clones, therefore an ANN with an 8-fold symmetries will be generated. The non-terminal <nn> is rewritten between 1 and 6 times, and generates a sub-ANN having between one and seven units. Because of the preceding 3 clones, this sub-ANN will be duplicated 8 times, however, one of those 8 sub-ANNs can potentially more than one leg.

The clone division and the normal cell division are specified in a similar way. The general type of the division is to be chosen between FULL, PAR, SEQ, CPI and CPO. Right after dividing, the two cells execute a different sequence of cutting operators generated with the <op> non-terminal. It is possible to adjust the probability of cutting link by first specifying how often we want to cut, and then how much links of a given type we want to cut, using the non-terminal <restrict>. We tune this probability so as to obtain ANNs not too densely connected. The second child sets the type of the newly created link between the two child cells, if there are some, using the non-terminal <settype>. When a cell stops to divide, it sets all its neuron attributes, the cellular code is generated by the non-terminal <unit>. First we reverse some links or/and add a recurrent link. We choose a particular the "amount of recurrence" by setting the probability with which recurrent links are created. <unit> then generates a time constant, some weights, a threshold, an initial activity, and finally the sigmoid type. The PI unit makes the product of its input. We use the PI unit with a small probability, because it is highly non-linear, and we do not want to introduce too much non-linearity.

7.3 Analysis of the second run

The `wavewalk` gait is when the robot moves one leg at a time, the quadripod gait is when the robot moves the legs four by four. For the second genetic run, we implemented a mechanism to store nice individuals, and be able to print them and film them afterwards. We now describe a selection of the different champions that were found during the second genetic run. They are shown in figure 0.11. At generation 0, the ANNs are a bit messy, densely connected. About 75 percents of the ANNs do not have the correct number of outputs, that is 8 outputs, they are directly eliminated. For this reason, the generation 0 takes quite more time than the other generations. Later the individuals have most of the time the right number of outputs. Among those who have the correct number of outputs, most of the ANN do not even make the robot move, also a lot of them make random movement, this is when a threshold unit is connected directly to the servos, its input is near to 0, therefore its output oscillates between 0 and 4096 due to the 1 percent of random noise that is added to the net input. Some ANNs produce a short sequence on one leg before the motionless state. They get a fitness like 0.001. One of them produced an oscillation on four legs. During the next two generations, we concentrated on evolving oscillations on as many legs as possible, giving fitnesses between 0.01 and 0.02, depending on how many legs were oscillating, and how good was the period and the duration of respectively the

```
<clone> [3..3];
<nn> [1..6];
<evolved> ::= <clone>
<coef2> ::= (SWEIGHTO: list[16..16] of (integer[0..+512]))
<clone>::=  (PROGN (FULL(<opi>)(<opo>)) (<clone>) | (PROGN (PAR(<opi>)(<opo>)) (<clone>))
           | (PROGN (SEQ(<opi>)(<opo>)) (<clone>)) | (PROGN (CPI(<opi>)(<opo>)) (<clone>))
           | PROGN (CPO(<opi>)(<opo>)) (<clone>)) |  (<nn>)

<nn> ::=   (FULL ( PROGN(<opi>)(<nn>)) (PROGN(<opo>)(<nn>)) )
         |  (PAR ( PROGN(<opi>)(<nn>)) (PROGN(<opo>)(<nn>)) )
         |  (SEQ ( PROGN(<opi>)(<nn>)) (PROGN(<opo>)(<nn>)) )
         |  (CPI ( PROGN(<opi>)(<nn>)) (PROGN(<opo>)(<nn>)) )
         |  (CPO ( PROGN(<opi>)(<nn>)) (PROGN(<opo>)(<nn>)) )
         |  (<unit>)
<opo> ::= (PROGN  (<op>)  ( <settype> )  (WAIT (integer[2..4]))  )
<opi> ::= (PROGN(<op>)(WAIT (integer[0..2])))
<op> ::=  (PROGN
           ( PROGN: set[0..7] of
                (PROGN(CTYPEI (0) )(<cuti>))  (PROGN(CTYPEI (1) )(<cuti>))
                (PROGN(CTYPEI (2) )(<cuti>))  (PROGN(CTYPEI (3) )(<cuti>))
                (PROGN(CTYPEI (4) )(<cuti>))  (PROGN(CTYPEI (5) )(<cuti>))
                (PROGN(CTYPEI (6) )(<cuti>))  (PROGN(CTYPEI (7) )(<cuti>)) )
           ( PROGN: set[0..7] of
                (PROGN(CTYPEO (0) )(<cuto>))  (PROGN(CTYPEO (1) )(<cuto>))
                (PROGN(CTYPEO (2) )(<cuto>))  (PROGN(CTYPEO (3) )(<cuto>))
                (PROGN(CTYPEO (4) )(<cuto>))  (PROGN(CTYPEO (5) )(<cuto>))
                (PROGN(CTYPEO (6) )(<cuto>))  (PROGN(CTYPEO (7) )(<cuto>)) ) )
<cuti> ::=  (PROGN (<restricti>)(RMI))
<cuto> ::=  (PROGN (<restricto>)(RMO))
<unit>  ::=   (PROGN
               ( PROGN : set[0..2]  of
                   ( WAIT ) (WAIT ) ( <cyc> )  ( PROGN: list[0..7] of (<reverse> ) ) )
               (PROGN  ( DELTAT (integer[1..40]))( <weight> )
    ( SBIAS  (integer[-4096..+4096])) ( SACT (integer[-4096..+4096]))  ) (<type>)  )
<reverse> ::=  (PROGN (CTYPEI(integer[1..7])) (<restricti>) (REVERSEI)  )
<command> ::= (PROGN
                (WAIT(200))(DELTAT(1)) (SBIAS(0)) (PROGN(CTYPEO (0) )(<coef2>))
                (PROGN(CTYPEO (1) )(<coef2>)) (LINEAR)   )
<weight> ::= (PROGN
                (PROGN(CTYPEI (2) )(<coef>))  (PROGN(CTYPEI (3) )(<coef>))
                (PROGN(CTYPEI (4) )(<coef>))  (PROGN(CTYPEI (5) )(<coef>))
                (PROGN(CTYPEI (6) )(<coef>))  (PROGN(CTYPEI (7) )(<coef>)) )
<coef> ::= (SWEIGHT: list[7..7] of (integer[-4096..+4096]))
           (SWEIGHT: list[7..7] of (integer[-256..+256]))
<restricti> ::= (RESTRICTI(integer[0..2])(integer[1..10]))
<restricto> ::= (RESTRICTO(integer[0..2])(integer[1..10]))
<type>  ::= (LINEAR)| (LINEAR)|(LINEAR)|(LINEAR)|(LINEAR)|(STEP)|(STEP)|(STEP)|(STEP)|(STEP)
            (PROGN(DELTAT(1))(PI))
<cyc> ::=  (PROGN (RESETNEW) (CYC) (<settype>)  )
<settype> ::= (PROGN (CNEWI)
               (PROGN : set[1..1] of
               (STYPEI(2)) (STYPEI(3)) (STYPEI(4))(STYPEI(5))(STYPEI(6)) (STYPEI(7))))
```

Figure 0.10: Syntactic constraints for the second run

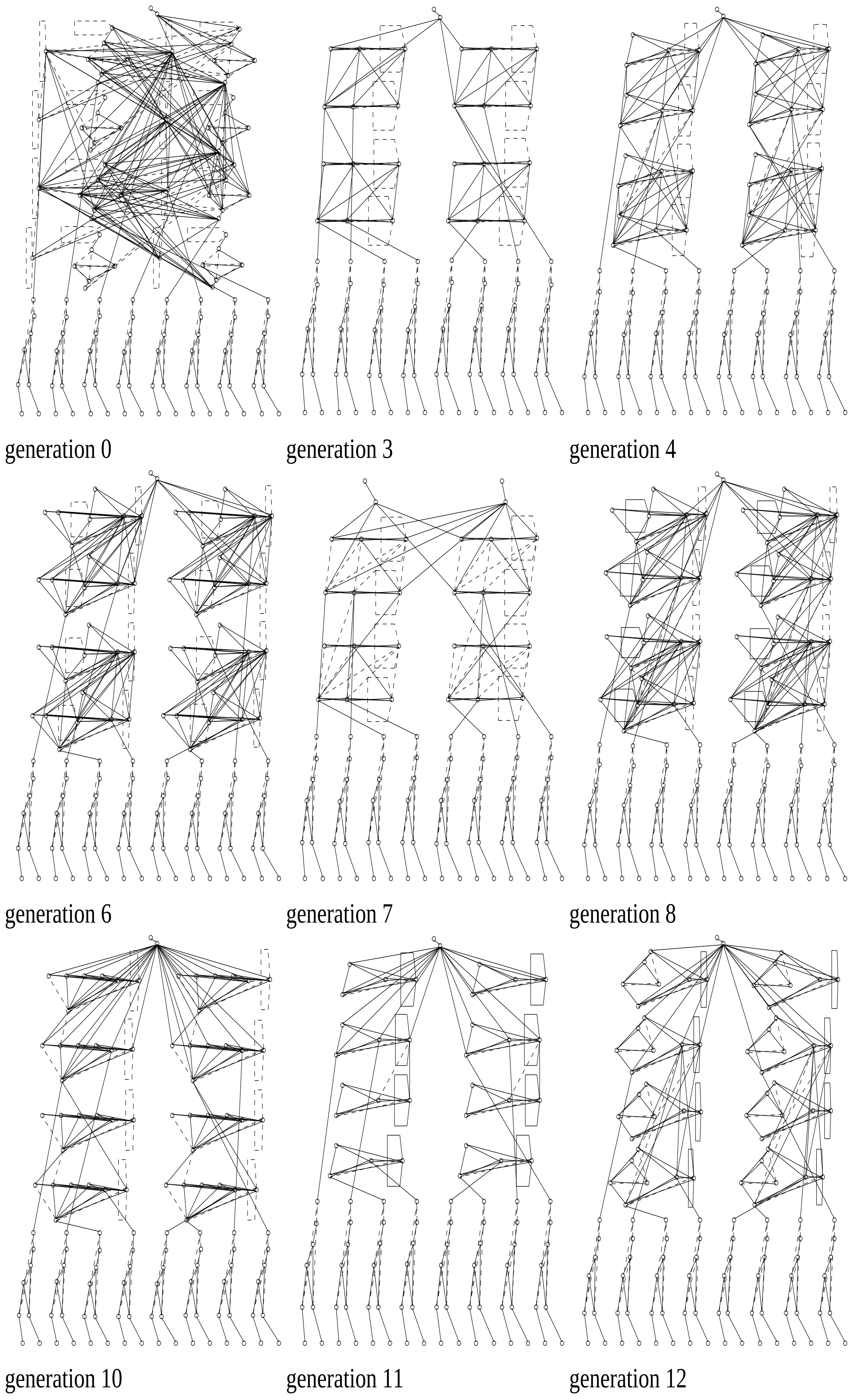

Figure 0.11: Champions of the second run

return stroke and the power stroke. At generation 3, we started to have individuals with a little bit of coordination between the legs. We watched an embryo of wavewalk which very soon vanished because the leg went out of synchrony. The coupling was to weak. The ANN is represented in figure 0.11 on the second picture, you can see that it is very sparsely connected. The next ANN at generation 4 generated an embryo of quadripod. Four legs were moved forward, then four other legs, then the 8 legs were moved backward. At generation 6 we got an ANN made of four completely separated sub-ANNs each one was controlling two legs, due to a different initial setting of the activities, the difference of phase was correct at the beginning, and we obtained perfect quadripod, but after 2 minutes, the movement decreased in amplitude, four legs come to a complete stop, and then the four other legs. Generation 7 gave an almost good quadripod, the phase between the two pairs of four legs was not yet correct. The ANN is probably a mutation of the guy at generation 3, because the architecture is very similar. Generation 8 gave a slow but safe quadripod walk. The coupling between the four sub-ANNs is strong enough to keep the delay in the phase of the oscillators. Only the frequency needs to be improved, because that's really too slow. Generation 10 produced a correct and fast quadripod, but not the fastest possible. Generation 11 produced a guy which walked a little bit faster, due to an adjustment of the phase between the two pairs of four legs. The frequency did not improve. Furthermore there was no coupling between the legs. Generation 12 did not improve, We show it because it has a funny feature. The input neuron was not used in all our experiment, it was to control the speed of the robot, but we could not yet go that far. However its activity was set to 2000, and what happen at generation 12 was that an ANN was produced that used this neuron to differentiate the phase between the two pairs of sub-ANNs that were controlling the two pairs of four legs.

Except at generation 0 where we always have a great variety of architectures, you can see that the general organization of the ANN remain similar throughout the whole run. It has 8 sub-ANNs, four of them control 8 legs, one for two contiguous legs, and can potentially ensure that two contiguous leg on one side are out of phase. The four other ANN are in-between, some times they are not used at all, as is the case in the champion of generation 11 and 12. The 8 Sub-ANNs taken separately have some recurrent links. However, if each Sub-Ann is modeled as one neuron, the architecture becomes feed-forward. The Sub-Anns that control the rear legs send information to all the other Sub-Anns. They are the two master oscillators which impose the rhythm. The other sub-ANNs, although they could suposedly run independently, have their phase locked by the master oscillators.. We realized that in this run, it was not possible to mutate the general architecture. without destroying the whole genetic code, because that would imply replacing a division by another. We think it explains the fact that the general architecture did not change. So we decided to change the syntactic constraints in the next run so as to make the mutation of the general architecture possible, and more thoroughly explore the space of general architectures.

7.4 Analysis of the third run

The third run lasted only 5 hours, because of a typing mistake in the first generation. We typed a high fitness on the keyboard, for a bad individual who kept reproducing all the time during the next three generation. Being unable to locate it and kill it, we had to kill the whole population, a bit like the story of the mad cow disease which recently happened in England.

7.5 Syntactic Constraints used in the fourth run

Our motivation to do another run was to enable mutation of the general structure. We also realized that they were still bugs in the syntactic constraints and we fixed them. The new syntactic constraints are shown in figure 0.12. To enable mutation of the general structure, instead of implementing <clone> as a recursive non-terminal, we put three occurrences of it, and use it in a non-recursive way. In this way, it is possible to mutate any of them, without having to regenerate the others. Second, we used only the FULL division for the general structure, and force the EA to entirely determine type by type, which link are inherited and which are not. This can be mutated independently, and may result in making possible " soft mutation" that modify a small part of the division. Whereas if we mutate a

```
<evolved> ::= ( PROGN(<clone>)(<clone>)(<clone>)(<nn>))
<clone>::=  (FULL(<opi>)(<opo>))
<opo> ::= (PROGN  (<op>) (<settype>) (WAIT (integer[0..4]))  )
<opi> ::= (PROGN(<op>)(WAIT (integer[14..18])))
<op> ::=  (PROGN
           ( PROGN: set[0..4] of
                (PROGN: list[1..2] of (PROGN(CTYPEI (2) )(<cuti>)) )
                (PROGN: list[1..2] of (PROGN(CTYPEI (3) )(<cuti>)) )
                (PROGN: list[1..2] of (PROGN(CTYPEI (4) )(<cuti>)) )
                (PROGN: list[1..2] of (PROGN(CTYPEI (5) )(<cuti>)) )   )
           ( PROGN: set[0..4] of
                (PROGN: list[1..2] of (PROGN(CTYPEO (2) )(<cuto>)) )
                (PROGN: list[1..2] of (PROGN(CTYPEO (3) )(<cuto>)) )
                (PROGN: list[1..2] of (PROGN(CTYPEO (4) )(<cuto>)) )
                (PROGN: list[1..2] of (PROGN(CTYPEO (5) )(<cuto>)) ) )  )
<unit>  ::=   (PROGN
               (PROGN : set[0..2]  of
                    (WAIT)(WAIT )(<cyc>)( PROGN: list[0..7] of (<reverse> )))
               (PROGN
            (DELTAT (integer[1..40])) ( <weight> ) ( SBIAS  (integer[-4096..+4096]))
    (SACT (integer[-4096..+4096])))
               (<input1>)(<input2>) (<output>)  (<type>)  (END)   )

<restricti> ::= (RESTRICTI(integer[0..2])(integer[1..32]))  (WAIT)
<restricto> ::= (RESTRICTO(integer[0..2])(integer[1..32]))   (WAIT)
<type>  ::= (LINEAR)(LINEAR)(LINEAR) (LINEAR) (LINEAR)
            (STEP)(STEP) (STEP) (STEP)  (STEP)  (PROGN(DELTAT(1))(PI))
<cyc> ::=  (PROGN  (RESETNEW) (CYC) (<settype>) )
<settype> ::= (PROGN
               (PROGN : set[1..1] of
                 (PROGN (CNEWI)(STYPEI(2))(CNEWO)(STYPEO(2)))
                 (PROGN (CNEWI)(STYPEI(3))(CNEWO)(STYPEO(3)))
                 (PROGN (CNEWI)(STYPEI(4))(CNEWO)(STYPEO(4)))
                 (PROGN (CNEWI)(STYPEI(5))(CNEWO)(STYPEO(5)))  )   )
<input1> ::=   (PROGN : set[1..1] of (WAIT)(PROGN(CTYPEI(0))(RMI))(PROGN  (CTYPEI(0))(RMI)))
<input2> ::=   (PROGN : set[1..1] of (WAIT)(PROGN(CTYPEI(1))(RMI))(PROGN(CTYPEI(1))(RMI)))
<output> ::=   (PROGN : set[1..1] of (WAIT)(PROGN(CTYPEO(0))(RMO))(PROGN (CTYPEO(0))(RMO)))
```

Figure 0.12: Syntactic constraints used during the fourth run, we report only the non-terminal that are rewritten in a different way, compared to the previous run.

division from CPI to PAR, for example, all the division has to be re-generated from scratch. (We remind the reader that now the division is also genetically determined using link types and cutting links selectively after the division). The goal of using only FULL was also to augment the probability of coupling between two sub-ANNs. Using only FULL augments the density of connections, so we augmented the probability of cutting links to keep the density at the same level. Also we made a distinction between cutting links that were created during the development, and cutting the link to the output unit which is now done at the end of the development, by the non-terminal <output>, we felt it was better to encode individually for each neuron if the neuron is an input or an output unit. Those two modifications resulted in producing ANNs whose number of input was always a multiple of 8, and each of the sub ANN is now forced to control one and exactly one leg, unlike the preceding run. The last modification we did were to ensure that all the newly created link got a type. When a cell divides, the first child cell waits 14 extra time steps before dividing again. It make sure that the second child cell has time to set the types of the newly created links. Then each time a link is reversed or a recurrent link is added, we also make sure that the type is set, where we had forgotten in the previous constraints.

7.6 Analysis of the fourth run

A selection of the champions of this run are reported in figure 0.13. We deleted systematicaly ANNs that do not have the right number of inputs, or produced no movement at all, as in the second run. We represent at generation 0 a guy which produced oscillation on one leg. At generation one, we had an individual that has a correct oscillation on all the legs. That guy has a quite simple 16 neuron controller made of 8 oscillator neurons with a recurrent connection, and 8 neurons that implement a weak coupling. The oscillators loose synchronisation. In generation 1, we had an ANN that produced oscillation, and coupling between the two sides but not within one side. That means that the two front legs for example, or the two right legs are synchronous, but not the left front leg with the left rear leg. Generation 2 produced a fun individual which moved the two front legs two times quicker than the other 6 legs. You can see figure 0.13 fourth picture, that the ANN that controls the two front legs is much more sparsely connected than the one which controls the other 6 legs. Generation 2 produced another champion: a slow but correct quadripod gait. The genotype is probably a mutation of the second guy at generation 1, because the architectures are very similar. There is one sub-ANN for each leg, as is specified by the syntactic constraints, the sub-ANNs within one side are coupled, but the two sides are independent, as is clearly shown in the picture. Generation 3 produced a funny gait, with four pair of two coupled oscillators, inside each pair, one oscillator has the double frequency of the other, due to the coupling. The figure clearly shows the structure of the ANN. Generation 5 produced a quadripod gait, not too slow, but there there still lacks some coupling between the ANNs controlling the legs of one side. There are diagonal connections between the four groups, which implement coupling between the two sides, but there are no connections from top to bottom. At generation 6 we had a nice ANN with all the connection needed for coupling, but the frequency on the right side was slightly greater than on the left side, as a result the robot was constantly turning right. We would have thought that a higher frequency result in turning in the opposite direction, but the reverse is true at least for those particular frequencies which were almost similar. We got a number of individuals which were always turning right. Generation 7, we finally got success, a perfect quadripod, smooth and fast. We had the robot run for 20 minutes to check the phase lock.

8. Comparison of the Interactively Evolved Solution with the hand-coded solution

8.1 performance

We now want to compare this solution with the hand-coded solution. The hand-coded program is a wavewalk that has been done with a loop and a C program, by Koichi Ide, an engineer in Applied AI systems. In wavewalk, only one leg at a time is up. The evolved solution naturally produced a quadripod gait, where four legs are on the ground and four legs in the air. First we just look at the speed. The champion evolved robot does 7,5 cm/seconds. Whereas in all the previous run we could not do better than 15 cm/seconds. The period of leg update is 50 ms, The period of leg update in Koichi's controller was set to 64 ms, (4 cycles), and the speed is is 5cm/seconds , if the periode of the leg was 50 ms, we assume that the robot woud do 6.4 cm per seconds, so our solution walks faster with the same rythme on the leg impulses.

We used some constraints on the leg movement. The legs had been restricted to move between -96 and +96 to avoid clutch, and they cannot move by more than 40 to avoid motor warming up. There are no such constraints in Koichi's controller. It can be said that in Koichi's controller, the beta angle which control whether the leg is backward or forward, sometimes makes an increase of 96 in a single time step, so the genetic controller is more smooth.

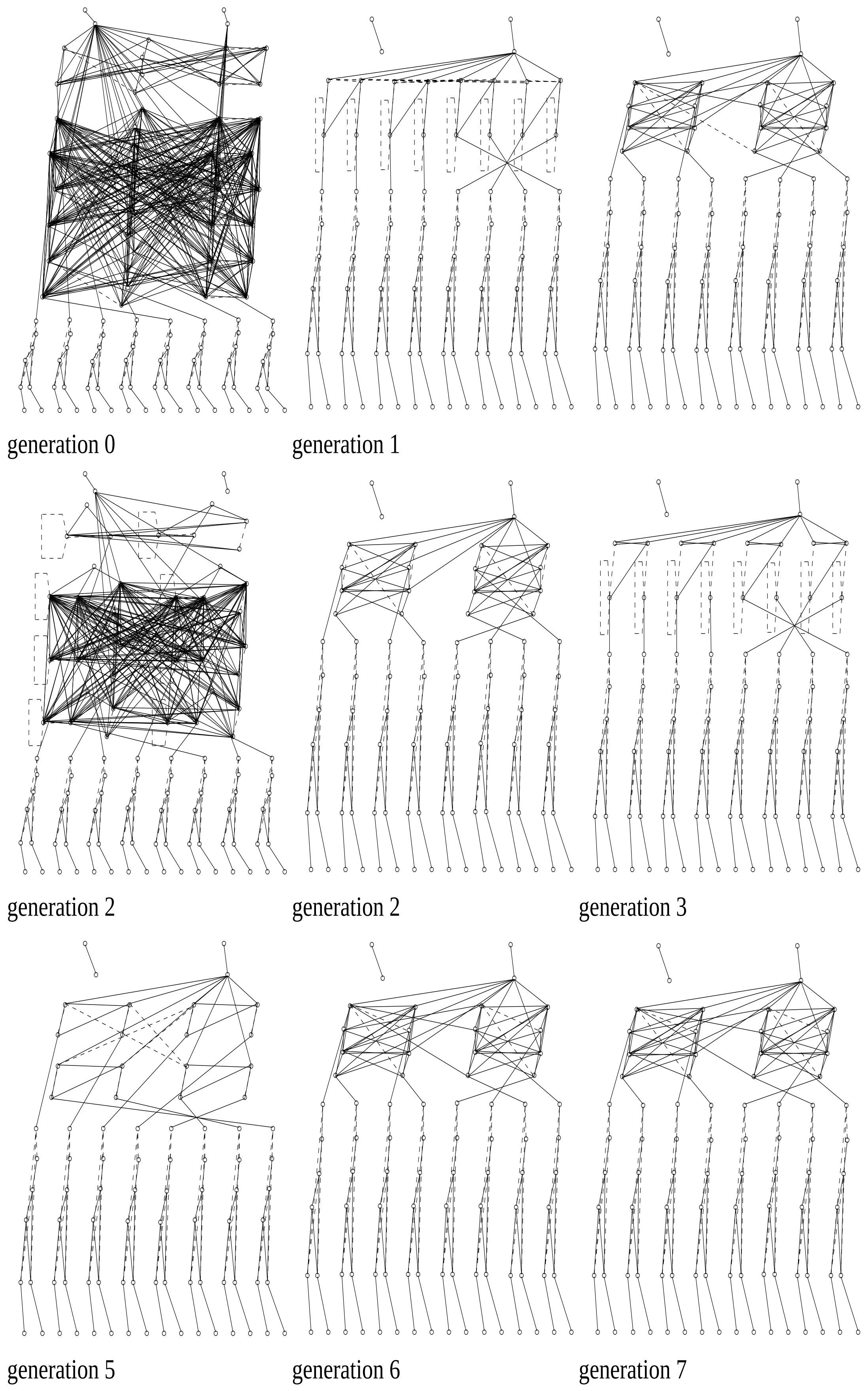

Figure 0.13: Champions of the fourth run

```
 LABEL(SEQ(SEQ(PAR(PROGN(WAIT(200 ))(DELTAT(1 ))(SBIAS(0 ))(PROGN(CTYPEO(0 ))
(SWEIGHTO(318 )(319 )(128 )(148 )(485 )(228 )(154 )(49 )(333 )(7 )(357 )(327 )
(314 )(444 )(171 )(448 )))(PROGN(CTYPEO(1 ))(SWEIGHTO(268 )(185 )(424 )(113 )
(54 )(357 )(316 )(110 )(259 )(102 )(90 )(43 )(299 )(367 )(477 )(78 )))(LINEAR)
)(PROGN(WAIT(200 ))(DELTAT(1 ))(SBIAS(0 ))(PROGN(CTYPEO(0 ))(SWEIGHTO(453 )
(461 )(82 )(56 )(283 )(111 )(385 )(43 )(409 )(312 )(391 )(210 )(491 )(347 )
(171 )(238 )))(PROGN(CTYPEO(1 ))(SWEIGHTO(67 )(403 )(483 )(458 )(104 )(219 )
(505 )(323 )(234 )(94 )(291 )(330 )(154 )(198 )(355 )(324 )))(LINEAR)))
(PROGN(WAIT(4 ))(CTYPEI(-1 ))(RESTRICTI(0 )(1 ))(STYPEI(0 ))(CTYPEI(-1 ))
(STYPEI(1 ))(CTYPEO(-1 ))(STYPEO(0 ))(FULL(PROGN(PROGN(PROGN(PROGN(PROGN
(CTYPEI(2 ))(PROGN(RESTRICTI(2 )(18 ))(RMI)))(PROGN(CTYPEI(2 ))(PROGN(WAIT)
(RMI))))(PROGN(PROGN(CTYPEI(4 ))(PROGN(WAIT)(RMI)))))(PROGN(PROGN(PROGN
(CTYPEO(3 ))(PROGN(WAIT)(RMO)))(PROGN(CTYPEO(3 ))(PROGN(RESTRICTO(0 )(17 ))
(RMO))))(PROGN(PROGN(CTYPEO(5 ))(PROGN(WAIT)(RMO))))))(WAIT(18 )))(PROGN
(PROGN(PROGN(PROGN(PROGN(CTYPEI(3 ))(PROGN(WAIT)(RMI))))(PROGN(PROGN
(CTYPEI(5 ))(PROGN(WAIT)(RMI)))(PROGN(CTYPEI(5 ))(PROGN(WAIT)(RMI)))))(PROGN)
)(PROGN(PROGN(PROGN(CNEWI)(STYPEI(2 ))(CNEWO)(STYPEO(2 )))))(WAIT(2 ))))
(FULL(PROGN(PROGN(PROGN(PROGN(PROGN(CTYPEI(5 ))(PROGN(RESTRICTI(2 )(16 ))
(RMI)))(PROGN(CTYPEI(5 ))(PROGN(WAIT)(RMI)))))(PROGN))(WAIT(14 )))(PROGN(
PROGN(PROGN(PROGN(PROGN(CTYPEI(2 ))(PROGN(RESTRICTI(1 )(3 ))(RMI)))(PROGN(
CTYPEI(2 ))(PROGN(WAIT)(RMI))))(PROGN(PROGN(CTYPEI(3 ))(PROGN(RESTRICTI(1 )
(31 ))(RMI)))(PROGN(CTYPEI(3 ))(PROGN(WAIT)(RMI))))(PROGN(PROGN(CTYPEI(4 ))
(PROGN(RESTRICTI(2 )(12 ))(RMI)))(PROGN(CTYPEI(4 ))(PROGN(WAIT)(RMI))))
(PROGN(PROGN(CTYPEI(5 ))(PROGN(WAIT)(RMI)))))(PROGN(PROGN(PROGN(CTYPEO(2 ))
(PROGN(WAIT)(RMO))))(PROGN(PROGN(CTYPEO(3 ))(PROGN(WAIT)(RMO)))(PROGN(CTYPEO(3
 ))(PROGN(WAIT)(RMO))))(PROGN(PROGN(CTYPEO(4 ))(PROGN(WAIT)(RMO)))(PROGN
(CTYPEO(4 ))(PROGN(RESTRICTO(0 )(9 ))(RMO))))(PROGN(PROGN(CTYPEO(5 ))(PROGN
(RESTRICTO(0 )(15 ))(RMO)))(PROGN(CTYPEO(5 ))(PROGN(RESTRICTO(1 )(21 ))
(RMO))))))(PROGN(PROGN(PROGN(CNEWI)(STYPEI(2 ))(CNEWO)(STYPEO(2 )))))
(WAIT(0 ))))(FULL(PROGN(PROGN(PROGN(PROGN(PROGN(CTYPEI(5 ))(PROGN(WAIT)
(RMI)))(PROGN(CTYPEI(5 ))(PROGN(WAIT)(RMI)))))(PROGN))(WAIT(16 )))(PROGN(
PROGN(PROGN(PROGN(PROGN(CTYPEI(2 ))(PROGN(WAIT)(RMI))))(PROGN(PROGN(CTYPEI(3 )
)(PROGN(WAIT)(RMI)))(PROGN(CTYPEI(3 ))(PROGN(RESTRICTI(2 )(6 ))(RMI))))
(PROGN(PROGN(CTYPEI(5 ))(PROGN(RESTRICTI(0 )(2 ))(RMI)))(PROGN(CTYPEI(5 ))
(PROGN(RESTRICTI(2 )(7 ))(RMI)))))(PROGN(PROGN(PROGN(CTYPEO(2 ))(PROGN
(RESTRICTO(1 )(2 ))(RMO))))(PROGN(PROGN(CTYPEO(3 ))(PROGN(RESTRICTO(2 )(9 ))
(RMO))))(PROGN(PROGN(CTYPEO(5 ))(PROGN(RESTRICTO(0 )(12 ))(RMO)))(PROGN(
CTYPEO(5 ))(PROGN(WAIT)(RMO))))))(PROGN(PROGN(PROGN(CNEWI)(STYPEI(4 ))(CNEWO)
(STYPEO(4 )))))(WAIT(2 ))))(CPO(PROGN(PROGN(PROGN(PROGN)(PROGN(PROGN(PROGN(
CTYPEO(2 ))(PROGN(WAIT)(RMO))))(PROGN(PROGN(CTYPEO(3 ))(PROGN(WAIT)(RMO))))
(PROGN(PROGN(CTYPEO(4 ))(PROGN(WAIT)(RMO)))(PROGN(CTYPEO(4 ))(PROGN(RESTRICTO
(0 )(6 ))(RMO))))(PROGN(PROGN(CTYPEO(5 ))(PROGN(WAIT)(RMO))))))(WAIT(18 )))
(PROGN(PROGN(PROGN(PROGN(CNEWI)(STYPEI(4 ))(CNEWO)(STYPEO(4 )))))(PROGN(PROGN
(WAIT))(PROGN(DELTAT(4 ))(PROGN(PROGN(CTYPEI(2 ))(SWEIGHT(254 )(212 )(170 )
(-63 )(-181 )(158 )(122 )))(PROGN(CTYPEI(3 ))(SWEIGHT(242 )(73 )(103 )(56 )
(9 )(226 )(48 )))(PROGN(CTYPEI(4 ))(SWEIGHT(4003 )(1411 )(-3628 )(3953 )
(1248 )(-1062 )(1202 )))(PROGN(CTYPEI(5 ))(SWEIGHT(-2642 )(-1926 )(2968 )
(-4094 )(-217 )(-577 )(-2340 ))))(SBIAS(1896 ))(SACT(989 )))(PROGN(PROGN
(CTYPEI(0 ))(RMI)))(PROGN(WAIT))(PROGN(PROGN(CTYPEO(0 ))(RMO)))(LINEAR)(END)
)))(PROGN(PROGN(PROGN(PROGN(PROGN(PROGN(CTYPEI(2 ))(PROGN(WAIT)(RMI))))
(PROGN(PROGN(CTYPEI(3 ))(PROGN(RESTRICTI(2 )(24 ))(RMI))))(PROGN(PROGN
(CTYPEI(4 ))(PROGN(WAIT)(RMI)))(PROGN(CTYPEI(4 ))(PROGN(WAIT)(RMI)))))(PROGN
(PROGN(PROGN(CTYPEO(2 ))(PROGN(RESTRICTO(1 )(24 ))(RMO))))(PROGN(PROGN(CTYPEO(
4 ))(PROGN(RESTRICTO(2 )(20 ))(RMO))))(PROGN(PROGN(CTYPEO(5 ))(PROGN(WAIT)
(RMO)))(PROGN(CTYPEO(5 ))(PROGN(RESTRICTO(2 )(9 ))(RMO))))))(PROGN(PROGN
(PROGN(CNEWI)(STYPEI(5 ))(CNEWO)(STYPEO(5 )))))(WAIT(3 )))(PROGN(PROGN(PROGN
(PROGN(CNEWI)(STYPEI(3 ))(CNEWO)(STYPEO(3 )))))(PROGN(PROGN)(PROGN(DELTAT
(27 ))(PROGN(PROGN(CTYPEI(2 ))(SWEIGHT(-3469 )(-1379 )(-3329 )(-2945 )(296 )
(-1376 )(-4096 )))(PROGN(CTYPEI(3 ))(SWEIGHT(196)(3165)(-1501 )(-3442)
(2994 )(-2912 )(1369 )))(PROGN(CTYPEI(4 ))(SWEIGHT(2875 )(-575 )(3329 )(-770 )
(-402 )(-793 )(3496 )))(PROGN(CTYPEI(5 ))(SWEIGHT(-118 )(-179 )(-25 )(-220 )
(110 )(-240 )(170 ))))(SBIAS(-1697 ))(SACT(-1746 )))(PROGN(WAIT))(PROGN(WAIT))
(PROGN(WAIT))(LINEAR)(END)))))))(PROGN(BLOC)(TESTIO8)(SHARI(JMP12)(SHARI
(JMP12)(SHARI(JMP12)(SHARI(JMP12)(PROGN(SWITCH)(SHARI(JMP12)(PROGN(SWITCH)
(SHARI(JMP12)(PROGN(SWITCH)(SHARI(JMP12)(JMP12)(1 )))(1 )))(1 )))(1 ))(1 ))(1 ))(1 ))))
```

Figure 0.14: The genetic code of the Champion

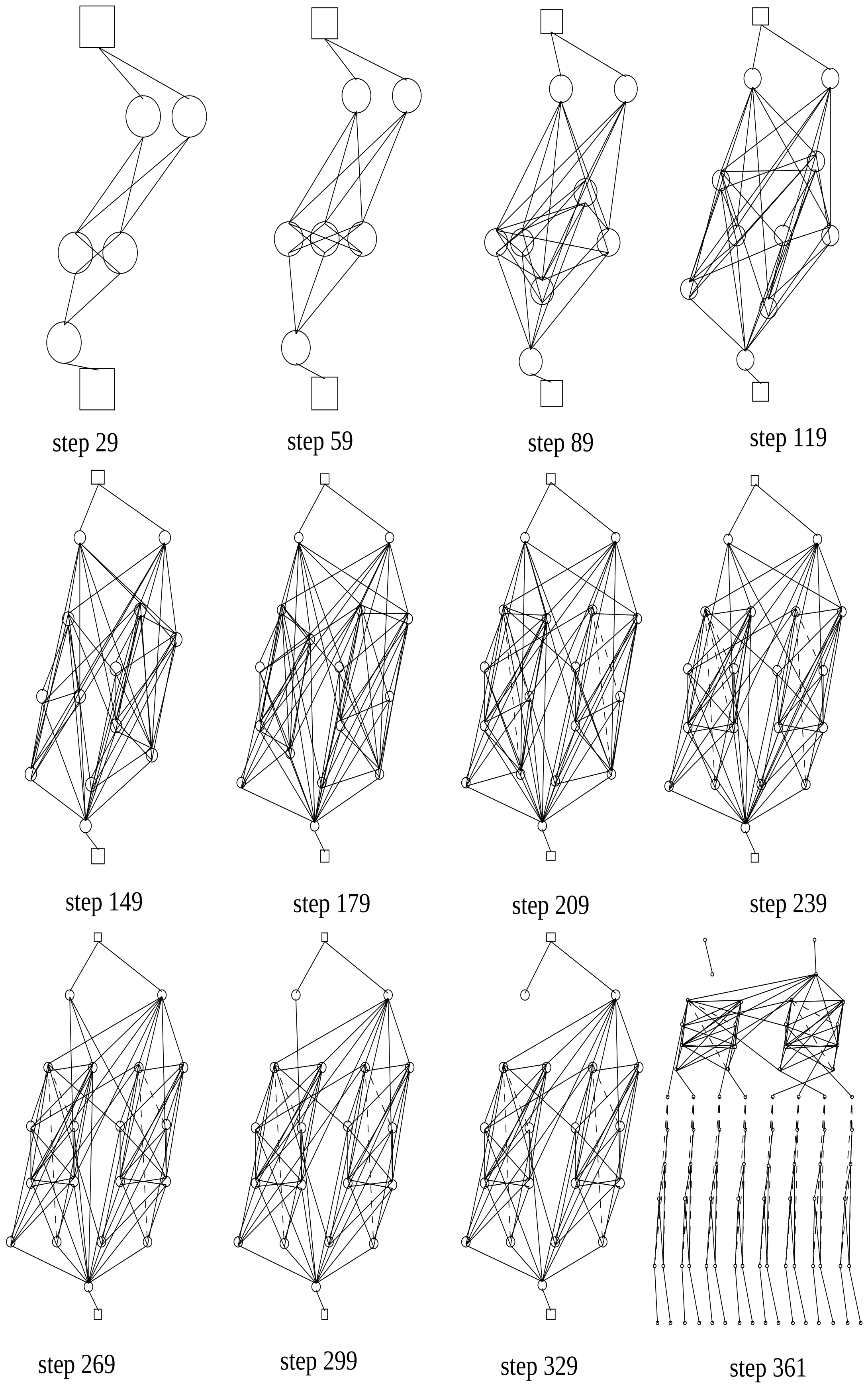

Figure 0.15: Steps of the development of the Champion

8.2 Analysis of the Solution

The genetic code of our best solution found in run 4 is shown in figure 0.14, and steps of its development are shown in figure 0.15. The genetic code is huge compared to the genetic code needed for one leg, that is shown in figure 0.6. This is because we need a lot of elementary cutting instruction, for each cell division, and also we go systematically through all the type to set the weights. By watching carefully the ANN, we found that it was only using LINEAR sigmoids, those are sigmoids which merely compute the identity. Puzzled by this fact we looked into the matter and remembered that when the net input is greater than 1,000,000 (resp. lower than -1,000,000) it is clipped to 1,000,000 (reps. -1,000,000). This is used to avoid computation artefact that could arise if the integer became too high. What the ANN does is that is that it exploits this fact by using big weights, and the identity function like a clipped linear sigmoid. Pure identity would always produce either a divergence to plus infinity, or a vanishing of the movement.

9. Conclusion

In this work we succeeded to evolve an ANN for controlling an 8-legged robot. Experiments were done without a simulator, and the fitness was determined interactively by hand. We believe that generating an ANN for locomotion controller is not a trivial task, because it needs at least 16 hidden units, unlike most applications that can be found in the literature. To be able to solve this problem in only 200 evaluations, four features were used.

- Using cell cloning and link typing.
- Interactive syntactic constraints.
- Interative problem decomposition.
- Interactive fitness.

Cell cloning generates highly symmetric ANNs. Link typing makes it possible to encode different type of cell division, and to genetically optimize the cell division itself. Syntactic constraints specify that there are exactly a three fold symmetry that exploit the fact that 8 legs are used, the problem would have been a little more difficult if only 6 legs were used. The hand given fitness allow us to reward behavior that would have been quite difficult to detect automatically.

Interactive operation of the Evolutionary Algorithm (EA) appears faster and easier than hand-programing the robot. It takes a few trial and errors if we are to generate the ANNs by hand. It is quite easy to say that the development must begin with three clone division, it is more difficult to say how exactly the division must be done. What's hand specified in our work is only the fact that there is a symmetry, not the precise architecture. The EA alone is able to decompose the problem of locomotion into the subproblem of generating an oscillator, and then makes 8 copies of the oscillator, and combine them with additional links so as to provide the adequate coupling. Other researchers have tried to evolved ANNs for legged-locomotion, but they all give more information than we do. In the work of Beer and Gallagher [2] the precise Beer architecture is given. This 6-legged architecture described by Beer in [1] has the shape of the number 8, in which all the sub-ANN controlling adjacent legs are connected, the controller of the legs that are symmetric between the two sides are also connected.

In the work of Lewis, Fagg and Solidum [10] not only is the general architecture given but also the precise wiring of all the neurons. This work is the first historic work where the fitness was given interactively by hand, and they also used a decomposition into sub-problem by hand. But the way it is done leaves the EA with only the task of finding four real values. They first generate a leg oscillator using 2 neurons, they only have to find 4 weights to obtain a correct rotation. Then they build the Beer archtitecture by connectiong 6 copies of this 2 neurons oscillator, with some precise links, and weight sharing between similar links. As a result, they once more have only four weights to genetically optimize.

We did not try to generate an ANN by hand, that could produce an efficient quadripod gait. I estimate the time needed to produce one to a few hours, one day for somebody who is not trained to think in terms of recurrent ANNs. We have not proved that the EA is a more efficient technique than direct hand programming, since it took two days. But, first, it is more fun to breed a robot during two days, than to try to understand the dynamic going on inside a recurrent ANN of a few dozens of neurons. Second, I could have asked my girlfriend to do the breeding, which would have reduced the "scientific time" down to less than what's needed to program the robot. Third, if we had have 10 robots to give the fitnesses then the time would have been reduced to two hours. This is a direction which we think is most interesting. The natural parallelism when using different robots is quite easily extendible. If you want to add a processor, you just need a bigger room to put your robot inside.

May be the most unexpected thing out of this work, is that the breeding is worth its pain. We are not sure that an automatic fitness evaluation that would have just measured the distance walked by the robot would had been successful, even in one week of simulation. There are some precise facts that support this view. First, right at the initial generation, we often got an individual which was just randomly moving its legs, but still managed to get forward quite a bit, using this trick. This is because the random signal has to go through the leg controller and the leg is up when it goes forward and down when it go backward. Using automatic fitness, the guy would have just dominate all the population right at generation 1, and all potentially interesting building blocks would have been lost. With interactive fitness we just systematically eradicate this noisy and useless individual. When we say noisy, it really do much more noise than all the others, because it moves all the time by the maximum allowable distance. So after a while, we push the kill button as soon as we hear the noise, kill, kill, kill, that gives an (un-healthy?) feeling of power. Second, there are some very nice features which do not result at all in making the robot go forward. We are pretty happy if we see at generation 1, a guy which move periodically a leg in the air, because that means that somewhere there is an oscillatory sub-structure that we would like to see spreading. Typically, we spend the first generations tracking oscillatory behavior, and tuning the frequency, we then rewards individuals who get the signal on all the height legs, and last, we evolve coupling between the legs, with the right phase delay. That's a pretty simple strategy to implement when breeding on line, but that would be dificult to program.

In short, we developed in this work a new paradigm for using Evolutionary Computation in an interactive way. Syntactic Constraints provide a prior probability (machine learning terminology) on the distribution of ANNs. Modular decomposition allow to replace one big problem by two simpler problems, Interactive fitness evaluation can steer the EA towards the solution.

Our future direction will be to evolve a locomotion controller with three command neurons: one for forward/backward, one for right/left and one for the walking speed. In order to do that, we need to enhance our method so has to be able to optimize different fitnesses with different populations, and then build one behavior out of two behavior separately evolved. In the case of turning, or speed controling things can be stated more precisely. We thing that turning as well as varying speed is only a matter of being able to govern the frequency of the oscillators. Typically we would like to evolve separately an oscillator whose frequency can be tuned using an input neuron, and then recombining it with the locomotion controller evolved in this paper. The way how to successfully operate recombination is still an open subject of research.

Acknowledgment

This paper reports on work partially done by Dominique Quatravaux in fullfillment of the master's thesis requirements under supervision of Michel Cosnard and Frédéric Gruau, on Frédéric Gruau's project at CWI in Amsterdam.

Since Dominique declined to be coauthor, it is proper to carefully delineate his contributions. He developed the gnu C compiler for the robot processor, and the protocol to handle the serial line, that is, to control the robot from the host computer. The programs can be downloaded from URL page `http://www.cwi.nl/ gruau/gruau/gruau.html`, together with user information, bugs and tips.

Dominique performed about half of the experiments (the leg controler, run0 and run1 of the locomotion controller) and developed the approach to first evolve a leg controler with a well defined interface. He can be reached at (quatrava@clipper.ens.fr). To represent Dominique's involvement we added the name of his workstations as co-author.

Phil Husband acted as a remote adviser, and we thank him for letting us use their OCT1 robot. Paul Vitanyi and the Algorithmics and Complexity (AA1) Group at CWI hosted the whole research, providing a modern computer environement and tools. We are grateful to Paul for his constructice and positive comments. F. Gruau was suppported by a postdoctoral TMR grant from the European community. within the evolutianary robotics group at the COGS department in Sussex University. We also acknowledge help from the pole Rhone Alpes de Science Cognitive who supplied money to buy a prometheus Robot in the frame of a joint project with prof. Demongeot. We prefered to use the OCT1 Robot because it turned out we had the choice and the OCT1 Robot was 10 time more expensive, more reliable and faster. We did however use replacement part from the prometeus robot, namely the power supply. Applied AI system did a good job in sending us replacement part for the OCT1 robot, together with advices. We thank Ann Griffith and Koichi Ide.

References

1. Randall Beer. *Intelligence as adaptive behavior.* Academic Press, 1990.
2. Randall Beer and John Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1:92–122, 1992.
3. Dave Cliff, Inman Harvey, and Cliff Husband. Exploration in evolutionary robotics. *Adaptive Behavior*, 1:73–110, 1993.
4. F.Gruau. Artificial cellular development in optimization and compilation. In Sanchez and Tomassini, editors, *Towards Evolvable Hardware.* Springer Verlag, LNCS, 1996.
5. F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD Thesis, Ecole Normale Supérieure de Lyon, 1994. ftp: lip.ens-lyon.fr pub/Rapports/PhD/PhD94-01-E.ps.Z (english) PhD94-01-F.ps.Z (french).
6. F. Gruau. Automatic definition of modular neural networks. *Adaptive Behavior V3N2*, pages 151–183, 1995.
7. Inman Harvey. Species adaptation genetic algorithm: a basis for continuing saga. Cogs csrp 221, The Evolutionary Robotics Lab, 1995. http://www.cogs.susx.ac.uk/lab/adapt/easy_csrps.html.
8. J.Kodjabachian and J. Meyer. Development, learning and evolution in animates. In *PerAc'94.* IEEE computer society press, 1994. anonymous ftp at ftp.ens.fr, pub/reports/biologie/PerAc94.ps.Z.
9. John R. Koza. *Genetic programming: A paradigm for genetically breeding computer population of computer programs to solve problems.* MIT press, 1992.
10. Lewis, Fagg, and Solidum. Genetic programming aprroach to the construction of a neural network for control of a walking robot. In *Proceedings of the IEEE International Conference on Robotics and Automation, Nice, France*, 1993. http://www.usc.edu/dept/robotics/brochure/rodney.html.