Evol. Intel. (2008) 1:187–207
DOI 10.1007/s12065-008-0015-7

RESEARCH PAPER

MENNAG: a modular, regular and hierarchical encoding for neural-networks based on attribute grammars

Jean-Baptiste Mouret · Stéphane Doncieux

Received: 17 December 2007 / Revised: 30 March 2008 / Accepted: 25 July 2008 / Published online: 25 September 2008

Abstract Recent work in the evolutionary computation field suggests that the implementation of the principles of modularity (functional localization of functions), repetition (multiple use of the same sub-structure) and hierarchy (recursive composition of sub-structures) could improve the evolvability of complex systems. The generation of neural networks through evolutionary algorithms should in particular benefit from an adapted use of these notions. We have consequently developed modular encoding for neural networks based on attribute grammars (MENNAG), a new encoding designed to generate the structure of neural networks and parameters with evolutionary algorithms, while explicitly enabling these three above-mentioned principles. We expressed this encoding in the formalism of attribute grammars in order to facilitate understanding and future modifications. It has been tested on two preliminary benchmark problems: cart-pole control and robotic arm control, the latter being specifically designed to evaluate the repetition capabilities of an encoding. We compared MENNAG to a direct encoding, ModNet, NEAT, a multi-layer perceptron with a fixed structure and to reference controllers. Results show that MENNAG performs better than comparable encodings on both problems, suggesting a promising potential for future applications.

Keywords Modular neural-networks · Evolutionary algorithms · Evolutionary robotics · Attribute grammars

J.-B. Mouret (✉) · S. Doncieux
Université Pierre et Marie Curie, Paris 6, FRE 2507, ISIR,
4 place Jussieu, 75005 Paris, France
e-mail: jean-baptiste.mouret@isir.fr

1 Introduction

Engineers and software developers make great efforts to subdivide their creations into sub-entities which can be reused and combined. The similarity between this approach and biological evolution was famously emphasized by Simon [56] with the parable of the two watchmakers. The first craftsman constructed watches so that each part depended on the others. As a result, if he happened to be interrupted during assembly, the watches fell to pieces. The second watchmaker adopted a modular method by dividing the watch into several sub-parts. Simon then shows that the first one's chances of completing a watch are very slim indeed when compared to the second one's, so demonstrating that nature would have followed a similar path. Since the publication of this seminal paper, modularity has been widely recognized as an important feature of both living organisms [26, 61] and artifacts. Many authors suggest that a modular design could increase both the evolvability and scalability of evolved systems [61, 62]. In neuropsychology, double dissociation studies [60], in which task deficits are mapped to brain damages, largely suggest a modular organization of animals' brain. This hypothesis have been recently refined using functional magnetic resonance imagery (fMRI [1, 30]) to, for instance, analyze the object-vision pathway [2].

Recent papers in the evolutionary computation field [28, 33, 42] have clarified the *modularity* concept by associating it with *regularity* (sometimes called repetition) and *hierarchy*. Modularity is defined as the structural localization of function that enables a group of elements to be handled as a unit. In an evolutionary context, this is related to the building block hypothesis [27]. Regularity is the repetition of similar sub-parts in a structure, making it more compressible. Hierarchy is the recursive composition of

sub-structures. Current engineering practices, biological observations [1, 26, 61] and recent evolutionary algorithm experiments [28] support the hypothesis that these three features are required in order to evolve complex systems.

There is no doubt that neural networks constitute such complex systems. They have proved their efficiency in many situations but are confronted to scalability and evolvability problems which could be addressed with a modular approach. Evolutionary robotics provides numerous examples of such situations, for instance when several legs have to be moved in a similar and synchronized manner [38], or when the wings of an artificial bird have to be carefully controlled [49]. In this work as in many other instances, complex neuro-controllers with unknown optimal topologies are sought.

Though many papers have described methods to evolve modular neural networks [3, 5, 14, 18, 24, 54], it would seem that none of them proposes a process exhibiting the three previously mentioned features for arbitrary networks. In this article, we describe a new neural network encoding scheme called modular encoding for neural network based on attribute grammars (MENNAG) that allows genetic operators to manipulate modules, repeat them, and hierarchically combine them. While we have focused our work primarily on the evolution of neural networks, the encoding could be easily adapted to evolve any graph structure. Modules are explicitly modeled in the genotype, noticeably because the emergence of modularity is a large debate [6, 34, 43] as it does not emerge in many evolutionary simulations [4].

As previously mentioned, many papers describe methods to evolve the topology of neural networks. Unfortunately, the most efficient of these systems involve a large and complex source code (often not available) as well as many implementation details neither not described nor justified in the papers. This complexity prevents researchers to easily re-implement other encodings to confirm the presented results. Moreover, it makes difficult the exploration and evaluation of variants of the encodings to distinguish its crucial features from the small refinements. In a few words, it is currently difficult to base future work on the literature because of the lack of a solid basis. Hussain [31] made a step forward to the establishment of a formalism for neural network encodings by demonstrating how some published encodings can be formally expressed using attribute grammars [36], a formalism designed to specify both the syntax and the semantics of programming languages. Following Hussain's proposition, the encoding we propose in this article is based on abstract syntax trees (AST), similarly to [24, 44, 38], which are described and interpreted using an attribute grammar.

We assessed the performance of the proposed encoding by comparing results with three current methods from literature [14, 58, 64]; on the classic cartpole problem [16, 21, 22, 32, 47, 52, 58, 64], and on the control of a planar robotic arm, which requires repetition of the same structure. To our knowledge, only a few papers have attempted to benchmark the available methods to evolve neural networks [23, 25, 32, 55, 58].

This paper is organized in seven parts. First, we briefly review the literature concerning the evolution of neural networks. Next, we introduce the new encoding scheme followed by the associated operators. The fourth part describes the results of each of the compared methods on the cartpole problem. Then results concerning the robotic arm problem are presented. A short discussion and some ideas about future work xconclude this paper.

2 Evolving neural networks

As pointed out by Yao [65], evolution can be involved in the design of a neural network at three different levels: synaptic weights choice (or more generally parameters choice), topology design and learning rules selection. In this paper, we are primarily interested in methods that evolve both synaptic weights and neural network structure, without any restriction on the reachable topologies. Two kinds of encoding strategies have been explored [37]: direct and indirect encodings. With a direct encoding, there exists a bijection between genotype and phenotype so that each neuron and each connection appears independently in the genome. An indirect encoding is a more compact representation; for instance, using development rules. More recently a new kind of encoding has been proposed: implicit encodings. These encodings are based on an analogy with gene regulation networks. The genome is a sequence of characters where specific tokens delimit neurons' description [45, 46]. These encodings seem to facilitate the evolvability of neural networks [53], but their ability to generate modular networks has not been studied yet. Yao [65] and Cantu-Paz and Kamath [8] provide a large overview of the combination of neural networks and evolutionary algorithms in a broader context.

Many systems have been suggested to directly encode a neural network. Binary strings have been used to describe a connectivity matrix [48] or variable-length lists of neurons and connections [9]. Some other authors have used a sequence of parameters [52]. Mutation of these genomes can easily be defined in most cases, but cross-over is seldom used due to the complexity of combining graphs efficiently. NEAT [58] offers an original solution by the introduction of innovation numbers, markers uniquely attributed when a new connection is created during the evolutionary process. These integers are employed to exchange similar connections— because historically derived from the same mutation and consequently

supposed to share a common functionality—during the cross-over. Such numbers avoid the need to run complicated graph-matching algorithms. Moreover, these numbers can be used to compute a distance that is exploited to create ecological niches [12], aimed at preserving diversity. Although these direct encodings gave good results on simple problems, they cannot generate hierarchical regular and modular neural networks.

During biological development, the complex process that links DNA code to phenotype allows the reuse of the same gene in many contexts, and many pleiotropic effects. This compactness makes it possible to describe complex individuals with comparatively little information. Indirect encodings try to broadly copy this principle by using various generative representation schemes. The graph-generating grammar, introduced by Kitano [35], used rewriting rules on the connectivity matrix. This research was pursued by Gruau who defined the cellular encoding [24] in which a chromosome is a development program syntactically specified by a context-free grammar. Instructions in this language are neuron-centered: they divide neurons in different manners, like a mitosis, and create connections between them. The representation in the form of an AST allows us to employ the genetic operators defined for genetic programming [39], such as, for instance, sub-tree swapping. Drawing inspiration once again from genetic programming, Gruau added some primitive automatically defined functions (ADF). Nonetheless, as highlighted in [44], swapping two sub-trees in cellular encodings is not the equivalent of swapping two sub-networks in the phenotype, because the instructions are execution-order dependent. Moreover, Gruau does not provide any elegant way to specify synaptic weights. This led Luke and Spector to define a similarly inspired encoding, but edge-based instead of node-based. The same approach has been followed in [29], in which the authors explored body-brain evolution by combining L-Systems [41] and edge encoding. At the same time, SGOCE [38] extended cellular encoding by situating cells in a geometric substrate and in two or three dimensions, thus enabling development instructions to take account of the relative positions of neurons in the substrate.

The concept of geometric substrate has also been exploited in [7] where an axon growth process was proposed. The genotype is made up of a list of cells, each one of them containing parameters such as the length of the axon or the development angle. Vaario [17] relied on another development scheme on a plane by explicitly taking into account the environment. In this model, each cell observes its neighborhood to execute one or several production rules. More recently, D'Ambrosio and Stanley [10] and Gauci and Stanley [20] exploited NEAT to evolve compositional pattern producing networks (CPNNs), a development abstraction able to represent repetitive patterns in a Cartesian space. These CPNNs are used to compute the connectivity and the synaptic weights between neurons situated on a plane.

The tree structure and the substrate concept are implicitly modular but, to our knowledge, no experiments have been carried out to evaluate the modularity of the networks obtained using the encodings described above. Repetition has been explored by Gruau [24] using a predefined number of sub-structures. It seems that with these encodings, hierarchy has never been studied explicitly.

An explicit repetition mechanism could be applied in order to efficiently reuse a sub-network or to exchange it during the cross-over, as proposed in the ModNet encoding [15]. A chromosome is made up of a list of model–modules, a list of modules and a list of links between those modules. The list of modules is constituted of model–modules, directly encoded, which can thus be used several times in the same network. Modules have only one input and one output. The cross-over is defined as an exchange of model–modules between chromosomes. Modular NEAT [54] constitutes another recent attempt to explicitly use modules. Modules are sub-networks encoded with NEAT that evolve symbiotically with a population of blueprints which specify how to combine them. Since modules cannot contain other modules, hierarchy is not present and, as a consequence, the complexity of modules is arbitrarily chosen (a few neurons in both cases). So far, ModNet has been successfully used to evolve controllers for cart-poles, lenticular blimps [15], and flapping-wing animats [49]. ModularNEAT has been tested on a simplified board game, demonstrating substantially better performance than NEAT.

The implementation complexity of these methods seems to have limited empirical comparisons and the number of problems on which they have been evaluated. Grönroos [23] compared a direct encoding, the graph-generating grammar [35], to the substrate-based encoding proposed in [50]. Working on the encoder problem and a classification task, the author's conclusions were similar to Kitano's: that the best performances on the encoder problem were achieved when using the graph grammar. However, he obtained the best results for the classification task with a direct encoding. Furthermore, Siddiqi [55] concluded that a properly set up direct encoding can be as efficient as Kitano's encoding on the encoder. Cellular encoding has been compared to direct encoding with a fixed topology in relation to the cartpole problem (see Sect. 4) and its variants. Both encodings led to working results but cellular encoding required less knowledge from the user. In [58], the number of evaluations required by NEAT has been compared to the numerical results published in [24]. The authors concluded that NEAT required about 25 times fewer evaluations than cellular encoding to obtain similar

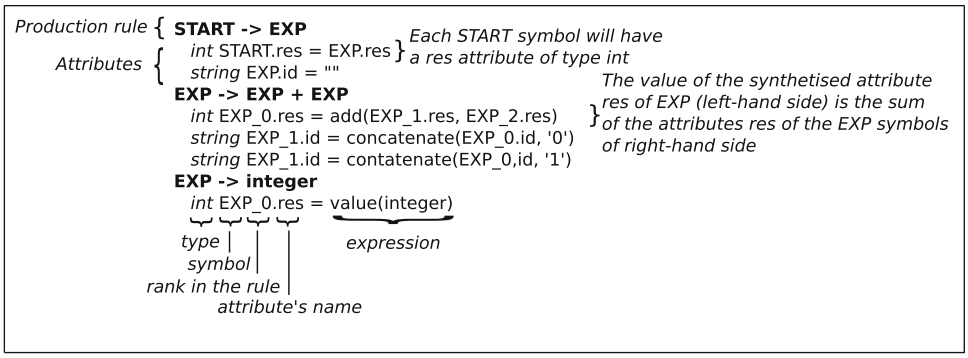

Fig. 1 Example of an attribute grammar which defines a language to add integers. The attributes *res* are synthesized, *id* are inherited. The result of the evaluation of programs is the value START.res. *id* attributes are only shown to illustrate inherited attributes behavior

results. Recently, Gomez et al. [22] compared several encoding schemes for neural networks on variants of the cartpole problem. The best results were obtained with topology-fixed methods, then by NEAT, cellular encoding ranking last.

This lack of precise comparison stems from the difficulty to implement most neural network encodings. The reasons are at least twofold: the complexity of the algorithms, which often have many parameters and an incompletely described inner working, especially in short articles, and the difficulty of implementation. Both problems highlight the need for a common formalism that might lead to generic tools, making the implementation and testing of proposed algorithms more straightforward. The formalism introduced by Hussain [31], based on attribute grammars [36], may fulfill this need. This extension of context-free grammars adds the capability to formally specify the semantics of a language. Each symbol is linked to a set of attributes and each production rule to a set of evaluation rules of these attributes (Figs. 1, 2). Attributes are divided into two distinct sets: *synthesized attributes*, which are the results of the update rules (and are computed from the terminals to the root symbol), and *inherited attributes*, whose value is issued from the parent node (and are consequently computed from the root to the terminals). Many algorithms have been developed to evaluate these attributes [51].

Individuals generation and development involves the three following steps:

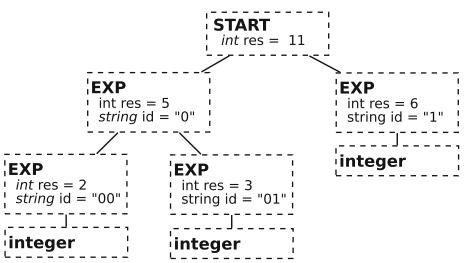

Fig. 2 Example of an abstract syntax tree for the expression "2 + 3 + 6", using the language defined by the attribute grammar of Fig. 1. Attributes are computed for each node, resulting in START.res = 11

- If the individual has to be randomly generated (at the beginning of the evolutionary process), the grammar makes it possible to generate a syntactically correct development program (a tree); if the individual is the result of a cross-over, the grammar is used to choose the crossing point without breaking the syntax.
- Once the tree has been generated, the attributes are evaluated.
- The list of connections and the list of neurons are two synthesized attributes of the start symbol used to build the neural network.

The operators used in genetic programming can be used to evolve programs described by an attribute grammar.

Hussain showed that a large class of neural network encodings can be formally expressed with these grammars, noticeably cellular encoding and its variants. Nonetheless, some encodings such as, for instance, NEAT, would probably be hard to express with such formalism because of their dependency on the evolutionary algorithm.

We implemented this in SFERES[1] [40], a software framework dedicated to evolutionary robotics experiments.

3 MENNAG: a modular encoding for neural networks

Let us first list our objectives in creating a new encoding scheme:

1. to encode all topologies (closure property);
2. to be at least as efficient as the current encoding schemes, particularly on problems of evolutionary robotics;
3. to enable modularity, hierarchy and regularity in neural-networks, in the hope that these properties will improve evolvability and be able to cope with more complex problems than do current encodings;
4. to implement an efficient way of exchanging sub-networks during cross-over;

[1] http://sferes.lip6.fr.

5. to be as formal as possible in order to allow easy re-implementation by other researchers and facilitate the exploration of variants.

ModNet and ModularNEAT enable modularity and repetition but are not formally specified and do not work with hierarchy of modules. Moreover, ModNet is limited to modules with only one input and one output. Cellular encoding and edge-encoding can be good inspiration sources, because they can be formalized using a context-free grammar, but their efficiency to allow repetition, modularity and regularity has not been demonstrated. The need to formalize and the possibility of exploiting tree-based genetic operators suggest using the framework described by Hussain. It is possible, with a correctly designed attribute-grammar, to ensure that exchanging a sub-tree corresponds to exchanging a sub-network.

Consequently, in this work we defined a new encoding scheme based on an attribute grammar that explicitly enables modularity, hierarchy and regularity. We called the new encoding MENNAG. Moreover, we had to create a new mutation operator, less destructive than the one used by Hussain.

Figure 3 outlines the different steps from MENNAG's attribute grammar to a modular, repetitive and hierarchical neural network. The genetic operators and the random generation of individuals rely on the MENNAG grammar to create syntactically correct parse trees (genotypes). The attributes are then evaluated to create a list of neurons and a list of connections, which is subsequently translated to a neural network. The following section details how the designed grammar defines individuals with the desired properties.

3.1 The mechanisms of the grammar

In this section, we present the main mechanisms used in our attribute grammar. The complete grammar can be consulted in the "Appendix". As most of the inner workings of the encoding are described by the grammar (except the mutation and cross-over operators), it appears quite complex at first glance, which is why we have chosen to present it step by step.

3.1.1 Neurons

The main structure of MENNAG's grammar relies on the non-terminals DIV and CELL, which respectively create a new module (DIV) and a new neuron (CELL), similarly to the Cellular Encoding. Each DIV level in the tree defines a module. These modules can be manipulated (exchanged, for instance) as independent units. An identifier, coded as a binary string, is given to each DIV and CELL as a function of the path within the tree linking it to the root node.[2] These identifiers are unique, and enable us to reference a module or a cell in order to create connections or duplicate a given module. The corresponding set of rules is depicted in Fig. 4. A sample syntax tree is shown in Fig. 4b.

3.1.2 Connections

Connections are made up of the identifiers of the neurons they connect and, contrary to Cellular Encoding, they have their own creation rules. The identifiers are represented as bitstrings, seen as constants by the attributes during the syntax tree interpretation but able to mutate during evolution, thanks to a bit-flip operator. To promote modularity, each DIV is associated with a list of connections that can only connect neurons created in the sub-trees of the DIV.[3] Synaptic weight is an attribute of each connection, mutated using Gaussian mutation. A connection is created using the rule in Fig. 5.

An overall direction can be given to connections to make networks more feed-forward. To that aim, it is easy to define rules that produce connections, for example, from the left sub-tree to the right. The rules are similar to the ones used for standard connections but "0" is added to the identifiers of the source neuron before concatenating it with the DIV's identifiers; and "1" to those of the target neuron. Thus, the source neuron is always in the left sub-tree and the target in the right, so creating an implicit direction in the computation flow from neurons created in the left part of the tree toward neurons created in the right. We have defined such a rule in our grammar, together with a simple rule without this constraint and tune selection probabilities (see sect. 3.2) so that these rules may control the aspect of the networks (see the complete grammar in "Appendix" for more details).

3.1.3 Inputs/outputs

We investigated two different ways to define input/output (I/O) connections: by creating a list of I/O attached to the root node, or by linking them to the leaves (i.e. to the neurons). In the first case, it is easy to ensure that each I/O is connected since the list is handled globally. But I/O do not belong to any module, and so cannot be exchanged during cross-over with the sub-graphs to which they are linked. Moreover, this approach prevents them from being correctly handled when a module is duplicated by a clone,

[2] For each DIV instruction, "0" is concatenated to its identifier to obtain that of its left son and "1" for its right one.

[3] To obtain this behavior, the identifier of the DIV is passed down to the CONN symbol. The DIV identifier is then concatenated to the evolved bit-strings representing source and target neurons, thus guaranteeing that these connected neurons belong to one the two sub-trees of the DIV.

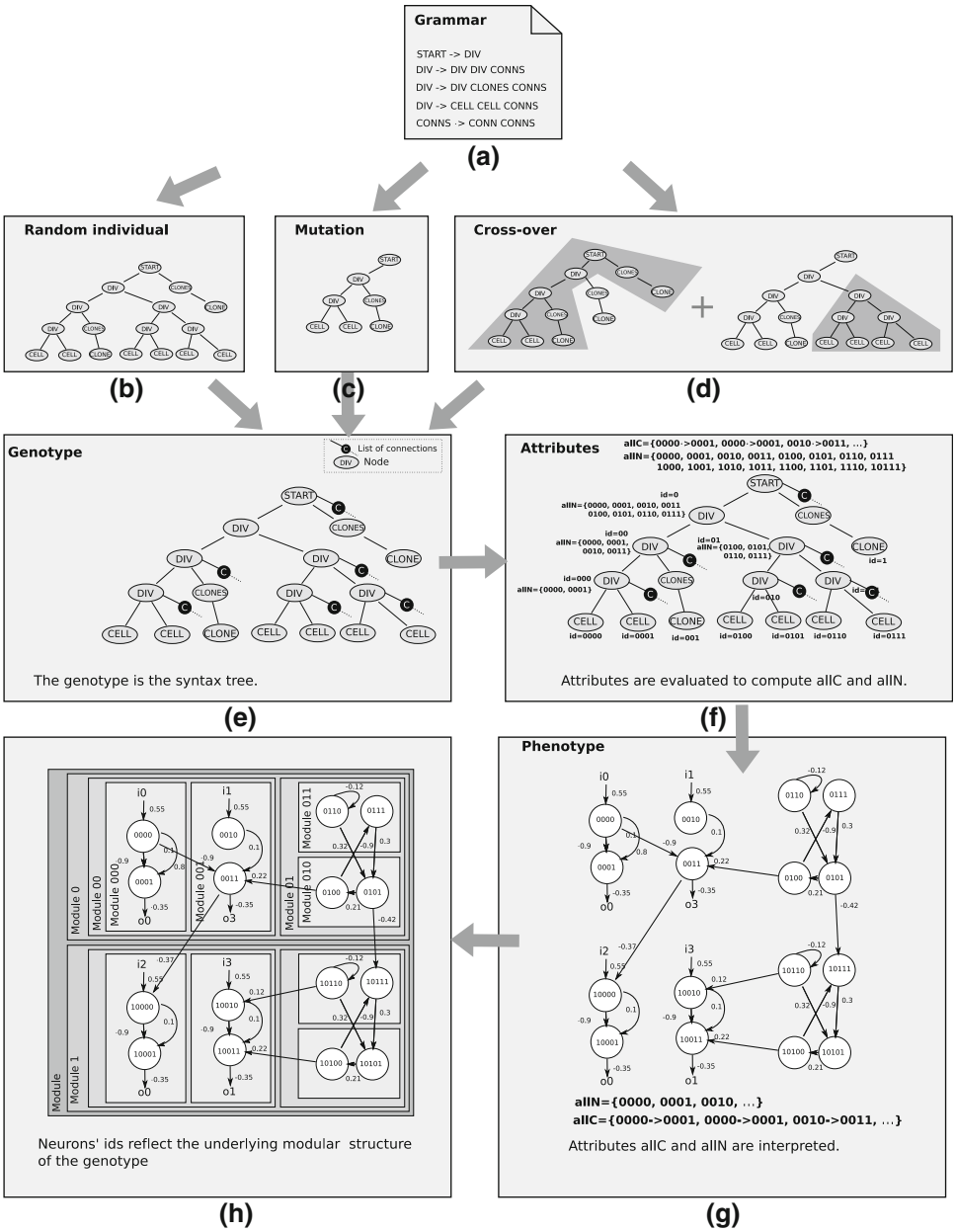

Fig. 3 The defined grammar (**a**) is used by genetic operators (**b**–**d**) to create new syntactically correct genotypes, defined as parse trees (**e**). Attributes are attached to each node of the tree, according to the grammar. The attributes START.allC (all connections) and START.allN (all neurons) are then evaluated (**f**). These attributes are easily transformed to a neural network (**g**) by scanning the two lists. Last, a modular view of the phenotype can be built using neurons' identifiers because they reflect the modular and hierarchical structure of the parse tree. Network (**h**) exhibits most of the presented features. Module 000 is repeated to define module 00. These modules are combined with two other modules to create the module 0. This module is itself repeated. A connection belonging to module 00 links a neuron from module 000 and one from module 001. Similarly, a connection created at the top level links neuron 0011 from module 0 to neuron 10000 from module 1. I/O are randomly swapped during each clone operation

(a)

DIV ⇒ DIV DIV CONNS
string CONNS.divId = DIV.id
string DIV_1.id = concatenate(DIV_0.id, "0")
string DIV_2.id = concatenate(DIV_0.id, "1")
set DIV.allC = union(DIV_1.allC, DIV_2.allC, CONNS.allC)
set DIV.allNodes = union(DIV_1.allNodes, DIV_2.allNodes)
DIV ⇒ CELL CELL
set DIV.allNodes = union(CELL_0.node, CELL_1.node)
string CELL_0.id = concatenate(DIV.id, "0")
string CELL_1.id = concatenate(DIV.id, "1")
CONNS ⇒ CONN CONNS
set CONNS.allC = union(CONNS_1.allC, CONN.allC)
CONNS ⇒ end
set CONNS.allC = createset()
CONN ⇒ end

(b)

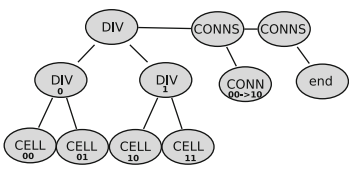

Fig. 4 **a** The main structure of the MENNAG grammar relies on a binary tree made of DIV instructions (modules) as node and of CELL instructions (neurons) as leaves. Functions used to update attributes are described in "Appendix C". The attributes of the CONN symbols are detailed in Fig. 5. **b** Example of a syntax tree which follows the grammar (**a**)

```
CONN ⇒ end
    evofloat weight.value = evolved
    bitstring targetNode.value = evolved
    bitstring sourceNode.value = evolved
    string CONN.targetNode = concatenate(CONN.targetNodeBaseId, target.value)
    string CONN.sourceNode = concatenate(CONN.sourceNodeBaseId, node.value)
    set CONN.allC = createset(createhash(name="sourceNode", value=CONN.sourceNode, name="targetNode",
value=CONN.targetNode, name="weight", value=weight.value)
```

Fig. 5 Each connection links two neurons, each identified by a bitstring which can mutate. To ensure that the source and target neurons are in the same sub-tree as the list of connections, the identifier of the DIV is concatenated at the beginning of the identifier of each neuron. Weights are encoded as real numbers (*evofloat*), able to mutate

as we will see later. On the other hand, if the I/O are attached to the neurons, the total number of used I/O is not available locally, which prevents the control of how I/O are used in the network. One possible solution might be to choose each I/O to be connected from a stack.[4] This approach has often been used in genetic programming. Gruau, for instance, used it to manage connection weights. The problem is that a mutation or a cross-over has many side effects, and may in particular imply changes in network parts not described by the modified sub-tree. In our case, this would mean that exchanging sub-trees would not correspond to exchanging sub-graphs. We then decided to link I/O connections to neurons, but without using a stack. The control of I/O connections is ensured by global constraints as we will see in Sect. 3.2.

To define I/O connections, we introduced the following production rules:

```
CELL ⇒ IN OUT
IN ⇒ end
IN ⇒ IO in_1
...
IN ⇒ IO in_k
OUT ⇒ end
OUT ⇒ IO out_1
...
OUT ⇒ IO out_l
```

for a network with k inputs and l outputs, in_i and out_i referring to input number i and, resp. output number i.

3.1.4 CLONE

The last important part of MENNAG grammar deals with the non-terminal CLONE, designed to duplicate sub-networks. A CLONE copies a module (DIV), which can contain itself one or more CLONE. We add the rules:

```
DIV ⇒ DIV CLONES CONNS
DIV ⇒ CLONES DIV CONNS
CLONES ⇒ CLONE CLONES
CLONES ⇒ CLONE
CLONE ⇒ end
```

[4] the first connection to input is connected to the first input, etc.

The principle of the CLONE consists in copying the list of neurons and the list of connections of the DIV, while changing the indices.

Two main issues must be resolved if we are to write the attribute updates corresponding to these rules: to ascribe new identifiers to the cloned neurons, while preserving the structure of the sub-graph and handling the I/O connections contained in the cloned module. A solution to the first issue consists in concatenating "1" (or "0" if the CLONE appears before the DIV) before identifiers of neurons and before target and source neuron bitstrings. Connections and neurons, once cloned, can thus be added to the synthesized attributes allC and allN of the DIV from the left-hand side of the production rule. These cloned connections and neurons are handled in the same way as any "standard" (non-cloned) ones.

Handling I/O connections means finding them among the connections to be cloned and swapping them with "equivalent" ones. Imagine that the same treatment (for instance, computing a derivative) might be applied to a set of I/O pairs. For instance, input 1 (I_1) should be derived to compute output 1 (O_1), input 2 (I_2) to compute output 2 (O_2), and so on. If the evolutionary process managed to create a module able to compute the derivative of input 1, it ought to be able to duplicate this module and replace I_1 by I_2. To obtain this behavior, we must find a way to evolve a permutation of the I/O for each copy.

These permutations are encoded as a vector of real numbers. If k is the rank of a real number in the vector (unsorted) and i its rank in the same vector in ascending order, then k should be permuted with i. For instance, in the vector [0.5;0.1;0.7], the rank of 0.1 is 1, the one of 0.5 is 2 and the one of 0.7 is 3. This vector could then be associated with [2;1;3] which is translated as "I_1 is replaced by I_2, I_2 by I_1 and I_3 is not changed". This simple way to encode a permutation allows us to exploit the genetic operators designed for real numbers optimization [11]. A vector attribute of evolved floats has thus been added to manage permutations of inputs and another one to manage permutation of outputs in the CLONE symbol (see detailed grammar in "Appendix").

The complete grammar is made up of the following production rules (see "Appendix" for the attributes):

```
START ⇒ DIV CONNS
DIV ⇒ DIV CLONES CONNS
DIV ⇒ CLONES DIV CONNS
CLONES ⇒ CLONE CLONES
CLONES ⇒ CLONE
DIV ⇒ DIV DIV CONNS
DIV ⇒ CELL CELL CONNS
CELL ⇒ IN OUT
CLONE ⇒ end
CONNS ⇒ CONN CONNS
CONNS ⇒ end
CONN ⇒ ICONN
CONN ⇒ ICONN
ICONN ⇒ weight node target
IN ⇒ end
IN ⇒ IO in1
IN ⇒ IO in2
OUT ⇒ end
OUT ⇒ IO out1
OUT ⇒ IO out2
IO ⇒ weight target
```

3.2 Genetic operators

Hussain [31] and Gruau [24] suggested the use of genetic programming operators [39]. However, we found that substantial improvements could be obtained by slightly modifying the initialization process and the mutation operators.

3.2.1 *Initial population (Fig. 3b)*

A selection probability is assigned to each production rule, enclosed in brackets within the grammar: for instance: [0.1] DIV ⇒ DIV DIV. In order to randomly create individuals, the basic algorithm begins with a starting symbol (START in our grammar) and creates a list of the applicable rules, similarly to the traditional GROW algorithm used in genetic programming [39]. The selection probability is then used to bias the choice of one production rule from the list of applicable rules. This process is run for each freshly created symbol until each syntax tree's branch ends with a terminal symbol.

Managing the properties of the trees created with this procedure is difficult. For instance, it can be very useful to create initial individuals with a specified number of neurons or with a particular mean number of connections. Furthermore, our local management of I/O connections makes it difficult to control how inputs and outputs are used. Though some theoretical analyses of the grammar and more powerful algorithms [19] may provide more elegant solutions, we chose a simpler method in this work: trees are generated using the procedure described above and those that do not follow certain constraints are removed. In our work, we limited the number of neurons and the number of connections by forcing them to remain within a given range, (see "Appendix" for typical chosen values). These constraints are fast to compute once the attributes have been evaluated. Noticeably, they do not require fitness evaluation.

3.2.2 *Mutation (Fig. 3c)*

Traditional genetic programming mutation relies on variants of the GROW algorithm applied to a sub-tree: a node is randomly chosen and the corresponding sub-tree is regenerated using the same algorithm as is used to create new individuals. This modification of the syntax tree is not translated into small changes of the neural-network. An ideal mutation should at least allow to add/remove a connection; and add/remove a module clone. On the tree side, the insertion of a module cloning symbol may be obtained by adding a DIV level using the rule DIV ⇒ DIV CLONES, for instance. In this example, let us add an identifier to each symbol to make clearer the correspondence before and after insertion. DIV_1 denotes the DIV we would like to clone. DIV_0, the DIV symbol on the left hand side, would be the father of DIV_1. A new DIV, DIV_3, would be inserted and DIV_1 would replace the DIV on the right-hand side of the inserted rule. The CLONES sub-tree would be generated using the GROW-like algorithm (Fig. 6). Corresponding grammar-rules are:

Before insertion:

```
...
DIV0 ⇒ DIV1 DIV2
...
```

After insertion:

```
...
DIV0 ⇒ DIV3 DIV2
DIV3 ⇒ DIV1 CLONES
CLONES ⇒ ...
...
```

By generalizing this idea, if an execution order-independent language is used then symbols can be safely added in the tree when there exists a recursive production rule to generate them. Such symbols include CONNS (CONNS ⇒ CONN CONNS) and DIV (DIV ⇒ DIV CLONES and DIV ⇒ DIV DIV).

A new mutation for the attribute grammar-specified genomes based on this insertion concept has been designed. To control its behavior precisely, two new numbers (i_1 and i_2) are enclosed in the brackets preceding production rules. They specify respectively the probability of selecting nodes

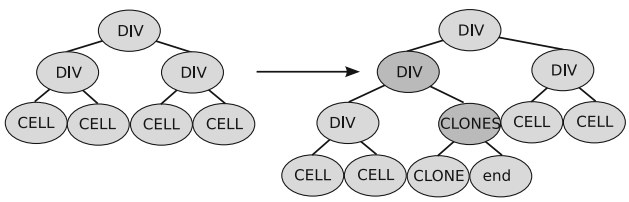

Fig. 6 Example of an insertion mutation

generated by this rule as insertion points and the probability to extend the tree using this rule. Here is an example of a full description of a production rule and the associated selection probabilities: [0.01, $i_1 = 0.5$, $i_2 = 0.5$] DIV $\Rightarrow$ DIV CLONES CONNS.

The corresponding mutation operator is implemented by executing the following procedure:

- randomly select an insertion rule r_1 used in the individual by biasing the choice using i_1;
- randomly select a recursive rule r_2 which will be used to extend the tree; the left hand side symbol of r_2 must belong to the right hand side symbols of r_1; the probability i_2 is used to bias the choice;
- among the nodes generated using r_1, randomly select a node n_1 with the same symbol as on the left hand side of r_2;
- create a new node n_2 with the same symbol as n_1;
- replace n_1 by n_2 in the list of sons of the father of n_1;
- use r_2 to randomly generate the sons of n_2;
- replace one of the sons of n_2 by n_1, in a compatible place.

A symmetrical mutation, which removes a node instead of adding one, can easily be derived from the same principles. This has been implemented in MENNAG. Moreover, the traditional genetic programming mutation can still be used with a low probability to create new subtrees.

4 Cart-pole experiment

This encoding was first benchmarked on the classical cart-pole problem (sometimes referred as pole balancing or inverted pendulum). This task has been chosen to check that MENNAG can solve a simple non-modular control task at least as efficiently as other encodings.

A pole, attached to a cart by a hinge, has to be balanced by moving the cart (Fig. 7). The goal is to maintain the pole in a vertical position while keeping the cart at the center of the track. This problem has been used to evaluate numerous encoding schemes for neuro-controllers [16, 21, 22, 32, 52, 58, 64] because it is representative of a wide class of control problems involving unstable systems. Contrary to some previous work, we do not consider the cart-pole problem as solved if the pole does not fall but only if the controller is able to keep the pole vertically without oscillating, as a classic proportional-derivative (PD) controller does. This prevents this task to be solved with a single neuron [63] and makes it more challenging. For instance, the path between the input and the outputs should be as short as possible to react as fast as possible.

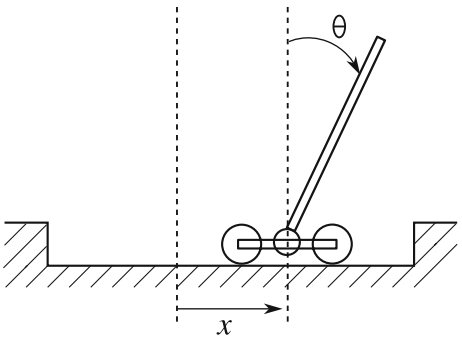

Fig. 7 Cart-pole problem. The pole has to be maintained in a vertical position while keeping the cart as close as possible to the center of the track

If the angle and the position of the cart are the only inputs, the controllers have to derive the input signals to avoid oscillating behaviors. Automatically discovering a neural-network able to perform such computation is the main challenge of this problem. In the following experiments, we did not provide the networks with velocities: pole angle and cart positions were the only inputs. The networks have a single output, which controls cart acceleration.[5] Although modular properties are not mandatory to solve the problem, it has been shown that they may help by allowing the propagation of a derivative module [15].

Equations enabling the simulation of an inverted pendulum are widely available (e.g., [13, 52, 64]).

These experiments were carried out with the fitness described in [15]. Let us first define the mean normalized error for the angle θ and the position x during the duration T:

$$e_\theta(g) = \frac{1}{T}\sum_{t=1}^{T} e_\theta(t,g)$$

$$e_x(g) = \frac{1}{T}\sum_{t=1}^{T} e_x(t,g)$$

where $e_\theta(t, g)$ and $e_x(t, g)$ denote the normalized errors at step t.

The fitness is the sum of two terms, a decimal and an integer:

$$f(g) = p(g) + \frac{1}{2}((1 - e_\theta(g)) + (1 - e_x(g)))$$

where $p(g)$ denotes the percentage of evaluation time the pendulum spent before going beyond the boundaries (± 0.2 rad and ± 2 m). This fitness, which varies between 0 and 101, ensures that controllers that maintain the cart-pole within the boundaries have a better fitness than the others, whatever the sum of errors may be.

[5] Neuron output belongs to the $[-1, 1]$ interval and is mapped to a force ranging between -10 and 10 N.

Many encodings have been tested on the double cartpole problem [58, 64], in which two poles are attached to the same cart. We did not benchmark MENNAG using this task because bootstrapping the evolutionary algorithm is especially difficult, making the task as much a problem of random generation and selection pressure than a problem of encoding. Moreover, our first tests with published encodings have shown that in many cases the evolutionary process was not able to fully solve the simple cart-pole task, being only able to prevent the pole from falling.

4.1 Results

We ran a set of 15 experiments for each encoding using the fitness previously described, a population of 100 individuals and a standard steady-state evolutionary algorithm. Figure 8 shows the obtained fitness with the MENNAG encoding, ModNet, a direct encoding similar to [52] and NEAT, using the original source code, (see "Appendix D" for detailed parameters used in this comparison. For reference, we added the fitness obtained using classic P and PD controllers tuned using a basic evolutionary search and those obtained with a simple multi-layer perceptron with one hidden layer (3 hidden neurons, 18 weights). An implementation of the cellular encoding based on the work of Hussain [31] was set up but we did not manage to get solutions with an efficiency at least similar to published results.

For all the encodings, the synaptic weights were encoded by a real number with Gaussian mutation.

Results are plotted on Fig. 8. NEAT obtains working controllers, with a fitness greater than 100, at the first generation. This is not surprising since it starts with a topology capable of emulating a P controller. While some MENNAG runs get good results in a few generations, about 60 generations are needed to obtain a fitness greater than 100 for all the runs. The most surprising result is the low performance of the multi-layer perceptron (MLP): despite the smaller search space, more generations are needed to achieve working controllers. This highlights an interesting case where evolving a topology is more efficient than using a fixed one. One of the main reasons may lie in the incremental path followed by evolution while evolving topologies. As a simple proportional link is sufficient to prevent the pole from falling, the individuals of the first generations use very simple networks with a few weights to tune. Evolution then improves behavior by adding neurons and connections. The multi-layer perceptron, on the contrary, starts with a complex and poorly-adapted topology with 18 weights to optimize simultaneously.

The results obtained with NEAT may be surprising given previously published results [58]. Although we tried to find the best parameters, we cannot claim that it would be impossible to improve the efficiency of NEAT on the investigated problem by using a different set of parameters. In [58], the authors investigated the cart-pole problem and their findings suggested than NEAT was very efficient in solving it. However, they did not focus their attention on fine control; they concentrated on the cartpole remaining in the boundaries and consequently "solved" the problem. In our benchmarks, NEAT led to working controllers in all runs, confirming the published results, but the more finely observed behavior was not as good as that obtained with a PD controller or MENNAG. We tried to use more generations but it did not improve the results.

All the encodings led to better fitness than the simple P controller but only MENNAG managed to produce at least one neural network as good as a PD controller, which prevents the pole from oscillating. Overall, MENNAG led

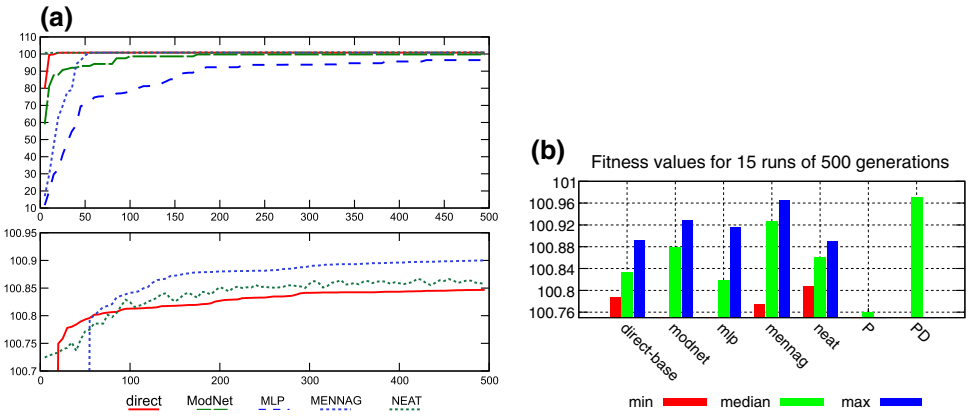

Fig. 8 **a** Mean fitness over 15 runs for the different encodings. **b** Min, median and max fitness values over 15 runs of 500 generations, for each encoding. No initial module was given to ModNet. P reference controller prevents the pole from falling, but cannot reduce the oscillations in the pole angle or in the cart position. PD reference controller quickly centers the cart with the pole in an upward position. All differences are statistically significant ($p < 0.05$, Wilcoxon–Mann–Whitney test ; see "Appendix A"), except between MENNAG and ModNet ($p = 0.074$)

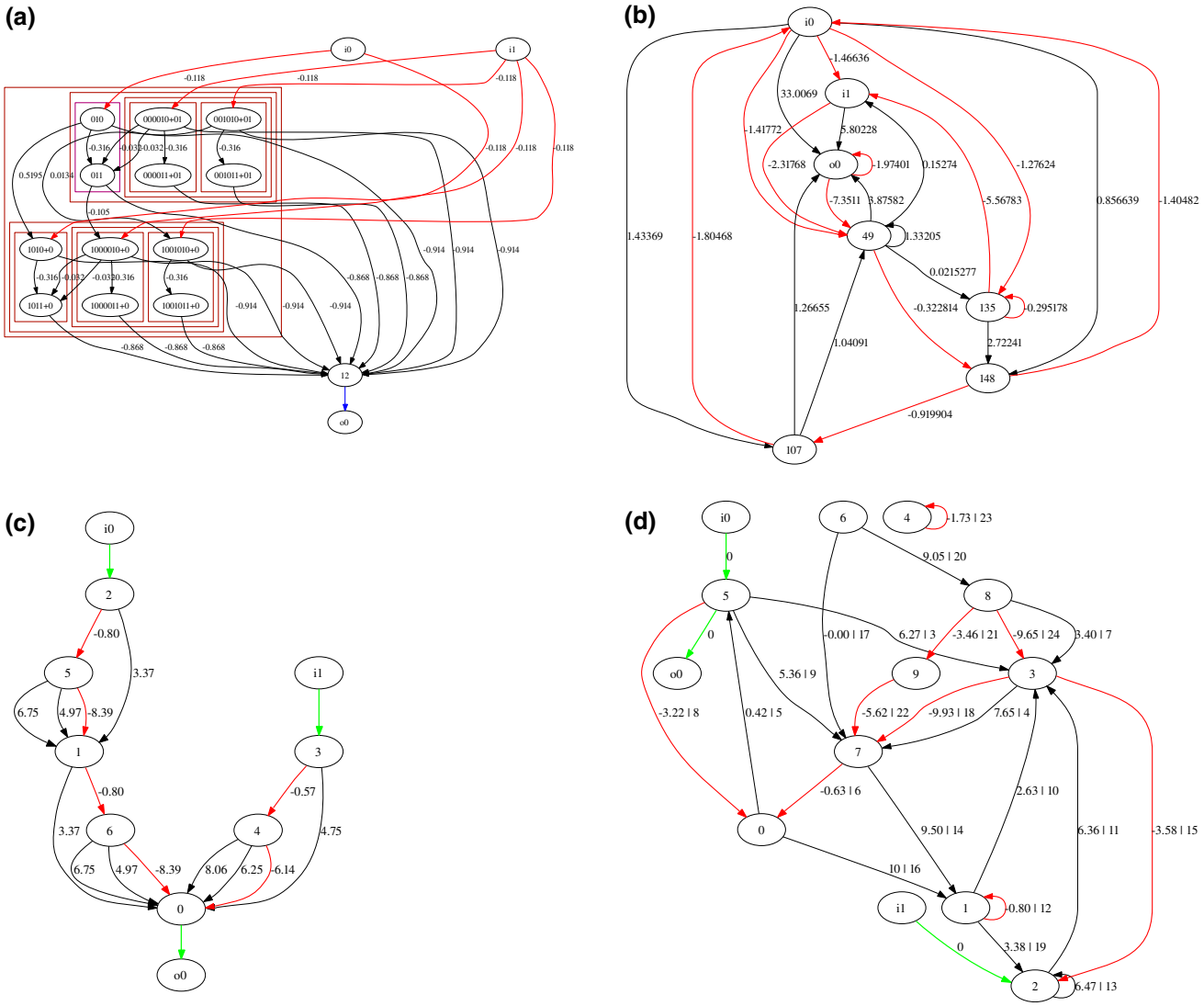

Fig. 9 Best neural networks after 1,500 generations with the compared encodings, for the cart-pole problem. i_0 is the angle input and i_1 the position input. o_0 is the network output. **a** MENNAG (fitness: 100.967); **b** NEAT (fitness:100.89); **c** ModNet (fitness: 100.93); **d** direct encoding (fitness: 100.899)

to significantly better results than the other methods. ModNet found some good controllers but certain runs failed even to generate a good P corrector.

Figure 9 depicts the best neural network obtained with each encoding. The genotype corresponding to the neural network obtained with MENNAG is seen in Fig. 10. Although this structure exploits the modular and hierarchical features of MENNAG, it has not created "derivative" modules and cloned them from one input to the other. Instead, it built the whole structure by cloning a simple module six times and by adding connections between the cloned modules.

MENNAG was able to generate several neuro-controllers as efficient as a PD-controller and more than half of the runs lead to better results than the best individuals obtained using the other encodings. Although the difference in fitness between the different solutions is small (a few hundredths), the qualitative difference on the behavior is not negligible: 100.858 corresponds to an oscillating behavior with an increasing amplitude, while 100.965 corresponds to a fast control without any oscillations at all (Fig. 11). At any rate, the reasons for this difference are unclear and require further study. Although we tried to tune the different encodings to obtain the best possible results, it is possible that some slight modifications reduce the gap between the solutions. The best neural networks generated with MENNAG use modularity, hierarchy and regularity, but it remains to be proved whether these features are effectively critical for this application, or

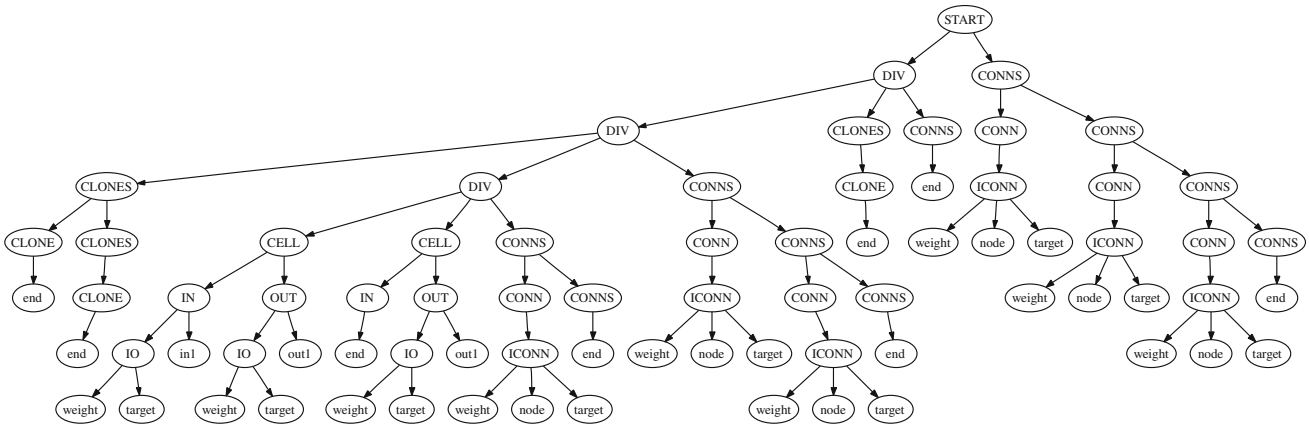

Fig. 10 Syntax tree that generates the neural-network of Fig. 9a

whether the good performances can be explained by other factors, like connections management, for instance.

We will now focus on another experiment in which the modular aspect of MENNAG should be even more decisive and more easily observed.

5 Robotic arm experiment

One of the main features of MENNAG is its ability to exploit the same sub-structure several times to solve similar sub-tasks. To evaluate the efficiency of this behavior, we designed a toy-problem based on the control of a simple robotic arm with three degrees of freedom (Fig. 12a) and simulated with the widely used dynamics library ODE.[6] The evolved neural networks have to drive each motor to a target position $T = (a_1,a_2,a_3)$ while knowing only T and its current position $P = (b_1, b_2, b_3)$. The motivation in the choice of this problem is twofold. First, the benefit of a repetitive encoding is obvious given the similarity between the degrees of freedoms. MENNAG should easily find good solutions whereas direct encodings are expected to be less efficient. Second, this problem is a simplified instance of many robotics tasks in which several degrees of freedom have to be controlled using similar strategies. Legged robots, in which each leg looks like the other ones, and industrial robot arms are simple examples.

The robotic arm problem can easily be solved by computing the difference between a_i and b_i ($i \in [1;3]$) to obtain the angular position error. This error can then be multiplied by a proportional factor to create a basic proportional controller (Fig. 12b). The same sub-network can be used to control each degree of freedom, making the problem completely decomposable. It should be noted that a network combining three times the structure of Fig. 12b should efficiently solve the problem. Such a network is a sparsely connected multi-layer perceptron and, consequently, a multi-layer perceptron should be able to control the arm.

This task is surprisingly hard for evolved neuro-controllers. For each degree of freedom, the algorithm has to:

- choose the right pair of inputs to connect to the right output;
- choose exactly opposite weights to compute a difference;
- find a proportional coefficient.

Most evolutionary algorithms for neural networks could easily find such a solution for one degree of freedom. However, if they are unable to repeat the structure previously found, they would have to find it three times over in the same network. The ability to manipulate modules in MENNAG should help to discover the 'difference' structure once and go on to clone it twice.

The chosen fitness is the normalized accumulation of errors between a_i and b_i, additioned for each degree of freedom:

$$F(g) = -\frac{1}{3T}\sum_{i=1}^{i=3}\sum_{t=0}^{t=T}|a_i(t) - b_i(t)|$$

where $i = 1, 2, 3$ and T denotes the number of time-steps. To ensure that the obtained controllers are able to generalize, five different sets of target position are used during each evaluation procedure.

5.1 Results

We ran a set of 15 experiments using the fitness described above, a population of 200 individuals and a standard steady-state evolutionary algorithm. A total of

[6] http://www.ode.org.

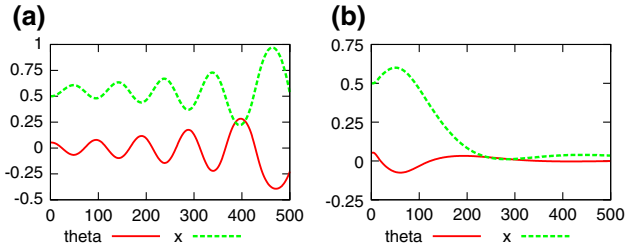

Fig. 11 Typical behaviors of the position x of the cart and of the value of ϑ for a 100.858 fitness (**a**) and a 100.965 fitness (**b**). At $t = 0$, the poles starts from $\vartheta = 0.01$ rad and $x = 0.5$ m. The controller has to drive the pole to $\vartheta = 0$ rad and $x = 0$ m. Controller (**a**) avoids the pole from falling during this evaluation, but the increasing amplitude suggests an imminent fall

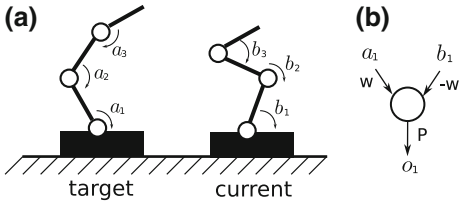

Fig. 12 **a** Robotic arm with three degrees of freedom. The controllers have to move each hinge to reach the target position as fast as possible, without oscillation. The robotic arm is simulated using a realistic dynamic simulator. **b** Example of a typical P controller for one degree of freedom

250 generations and the same parameters were used as in the former experiment, except concerning the number of I/Os.

Results are shown on Fig. 13. Among the investigated encodings, only MENNAG managed to reach the performance of a simple controller where each degree of freedom is connected to a P corrector. Moreover, the median values for the 15 runs show that these performances are obtained most of the time. The direct encoding and ModNet found 2-DOFs controllers while NEAT found no good P controller.

Some preliminary tests have been conducted with more degrees of freedom to understand how the proposed method scales up. While some good controllers have been found with four DOFs using MENNAG, the performance quickly decreases with more DOFs because of the combinatorial problem of selecting the good I/Os for each module. More details about this topic are available in the discussion section.

The evolution of the weights of a simple multi-layer perceptron with one hidden layer gave very good results and is given as a reference. While this performance may be surprising, it must be emphasized that the search space is much smaller when topologies are not explored. A brief analysis of the robotic arm easily led us to conclude that it should be efficiently controllable by a feed-forward neural network. However, by fixing the topology, we added a lot of information that was not available to the methods that were seeking efficient topologies.

Figure 14 is a sample of the best neural networks obtained with each encoding.

Figures 15, 17 and 16 detail one of the best controllers, obtained with MENNAG. A P-like structure has been found then repeated once using a CLONE instruction, demonstrating the usefulness of MENNAG's repetition instructions. This module is then duplicated using another CLONE instruction, at a higher level. In consequence, the same sub-structure is repeated four times (Fig. 16). Most successful networks observed did not succeed in obtaining precisely three instances of a P module but relied on a hierarchy of clones to obtain four instances. This behavior could possibly be improved by a finer tuning of the selection and insertion probabilities, but we opted to keep the same parameters for the two problems.

5.2 Lesion experiments

In order to evaluate the contribution of each component of the proposed system to the overall performance, we

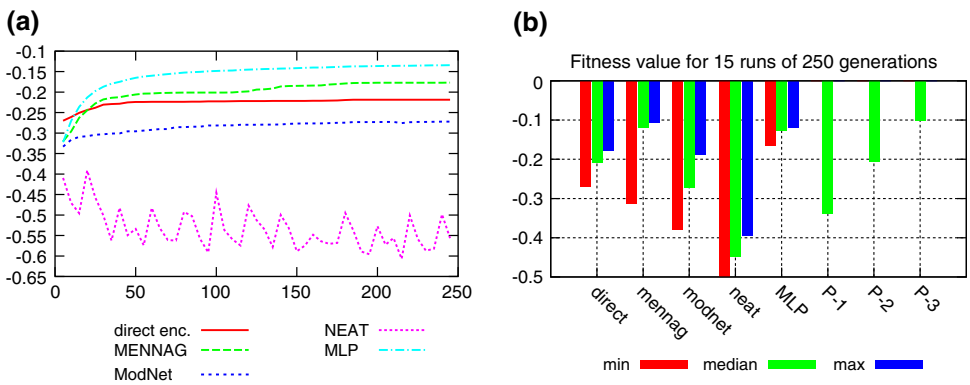

Fig. 13 **a** Mean fitness over 15 runs for the robotic arm experiment and the different encodings. **b** Fitness obtained after 250 generations. P-1 is the fitness obtained using one P controller, P-2 using two P controllers, P-3 using three P-controllers. All differences are statistically significant except between MENNAG and the MLP ($p \leq 0.02$, Wilcoxon–Mann–Whitney test; see "Table 2 in Appendix A")

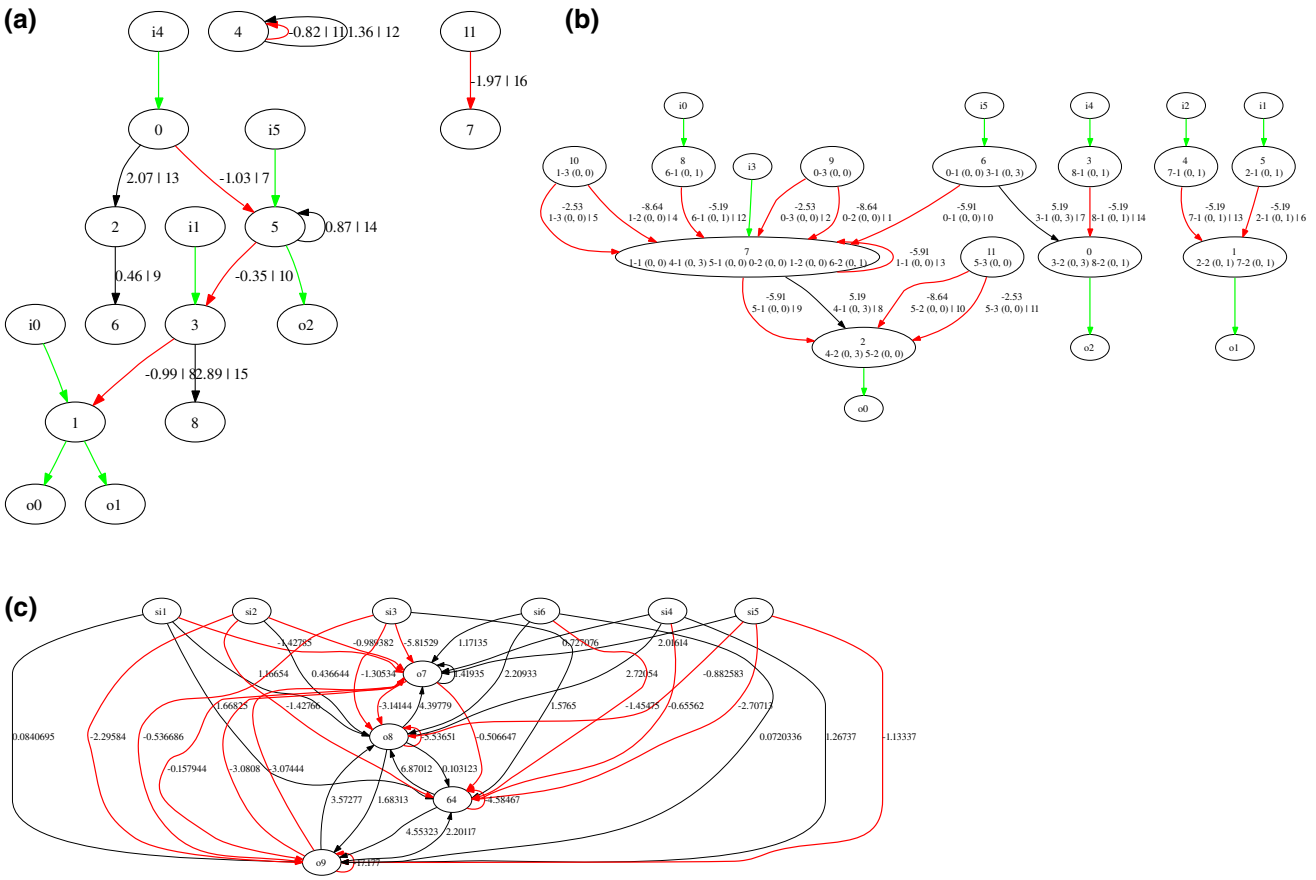

Fig. 14 Neural networks obtained with the different encoding on the robotic arm problem. **a** Direct encoding (fitness: −0.161155); **b** ModNet (fitness: −0.188119); **c** NEAT (fitness: −0.395931)

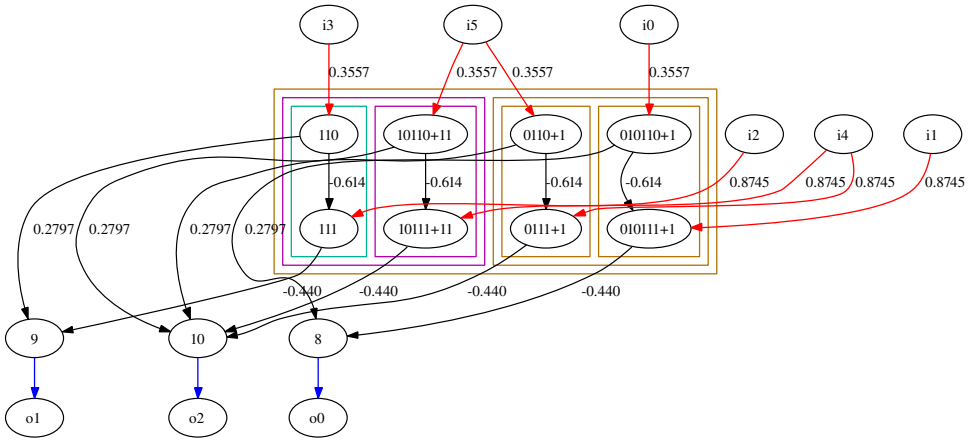

Fig. 15 Example of a 3-DOFs controller evolved using MENNAG with a fitness of −0.102298. Identifiers with a "+" symbols are used to track cloned modules. For instance, the identifier "10110 + 11" means that this neuron is part of a cloned version of the module "11". In this network, a P-like corrector connecting i2, i3 and o1 (module 11x, on the *left side*) has been cloned to link i5, i4 and o3 (module 1011x). These two modules have then been duplicated at level 1 to obtain the right side of the neural network

performed robotic arm experiments while successively removing each key part of MENNAG: CLONE instruction, insertion mutation and cross-over. Moreover, we tried to add the "random tree" mutation often used in genetic programming. This approach can be assimilated to the lesion experiments executed by biologists to understand living systems.

Results are plotted on Fig. 18. Best fitnesses were obtained with all the operators enabled excepting the random tree mutation. This mutation was used with a low

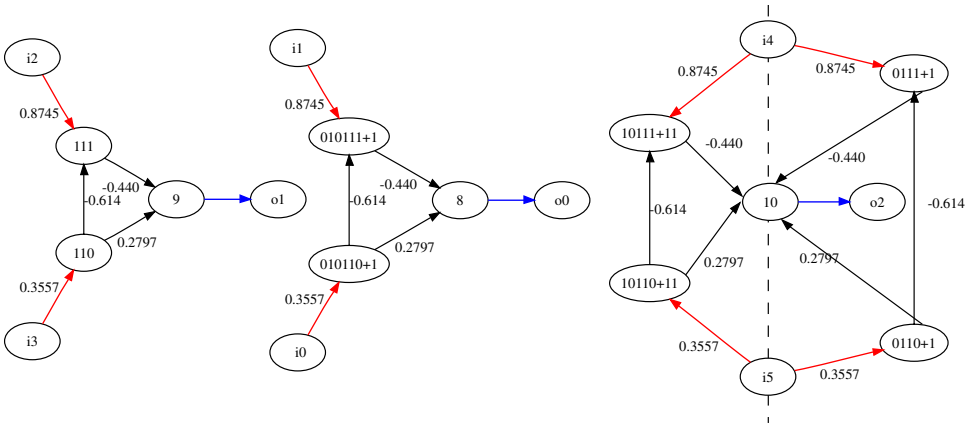

Fig. 16 Re-arrangement of the neural network depicted on Fig. 15 to obtain a better view of the repeated sub-structures. The *right* neural network has a vertical axis of symmetry. The two *left* networks differ only by their I/Os

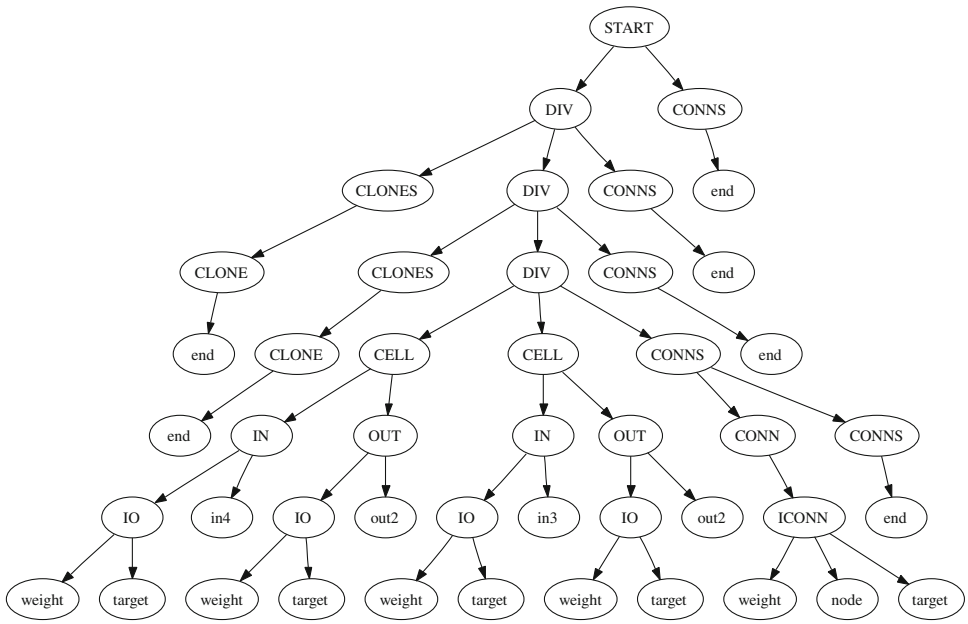

Fig. 17 Syntax tree (genome) that generates the neural network of Figs. 15 and 16

probability, so we did not expect it to have a strong influence on the results. Nevertheless, it does not seem to have improved them. Designed to maintain a diversity in the population, this operator may still be useful in other problems. The three degrees of freedom cannot be controlled if the tree's main components—insert mutation, CLONE and cross-over—are not used. If we consider the structure of the problem under investigation, the need for a CLONE operation seems obvious. The insertion mutation operator allows us to clone a module that has already been optimized, or at least efficiently tuned by evolution. CLONE and this operator are then complementary and reach their full potential when used simultaneously. Contrary to some other encodings that do not rely on cross-over, this operator proved to be useful in the proposed encoding. To know whether functional modules were exchanged, or whether cross-over was only used as a basic exploration method would require further investigations.

6 Discussion

This paper aims at two main contributions: proposing an efficient modular, regular and hierarchical encoding for neural network and expressing it using an appropriate and flexible formalism. Both ideas lead to good results but underline some difficulties.

MENNAG performed better than the comparable encodings we tested on the two simple tasks we investigated. More benchmarks are needed to gain a better view of what makes MENNAG more efficient. Modularity and the repetition features of MENNAG were demonstrated. Hierarchy was enabled in the presented experiments,

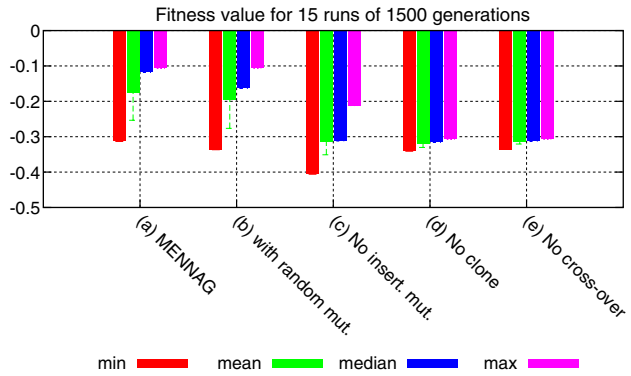

Fig. 18 Fitness obtained for the robotic arm experiment with different setups. *a* MENNAG; *b* MENNAG without the "random tree" mutation; *c* MENNAG without the new insertion mutation; *d* MENNAG without the CLONE instruction; *e* MENNAG without cross-over. Differences between *a* and *b* are not statistically significant (Wilcoxon–Mann–Whitney test; see "Table 3 in Appendix A") and differences between MENNAG and the other tests are significant ($p < 10^{-4}$)

although it is unclear whether it improved the evolvability of the neural networks. As pointed out in [62], designing a toy problem that would be both modular, hierarchical and repetitive is a difficult task which requires a deep understanding of modularity in complex systems. Moreover, some other kinds of modularity not easily available in MENNAG, such as repetition with variation and symmetry [59], may be required to evolve complex systems. The evolutionary robotics field could provide some interesting tasks as many of the envisioned situations imply showing different behaviors that intuitively require at least functional modularity in order to build up systems of increasing complexity while exploiting previously generated modules.

The proposed encoding used the attribute grammar formalism, proposed in the context of neural network evolution by Hussain [31]. This formalism interacts nicely with tree-based genotype and consequently with modular evolution, since trees are almost naturally modular and hierarchical. By choosing to use attribute grammars, our main goal was to ground future works on a solid basis by enabling an easy implementation (and future re-implementations) and to quickly explore a wide range of variants, for instance to benchmark different I/O handling strategies. Furthermore, the genetic operators are completely decoupled from the neural network problems, making possible to study them separately on simpler setups. The development of MENNAG proved us that experimenting with different alternatives was possible by only slightly altering the grammar, that is compact enough to be easily manipulated. This leaded to a faster development of the encoding. Moreover, as an illustration, we recently tried to re-implement MENNAG in a new attribute grammar system. Once the generic attribute grammar evaluator and the genetic operators were set-up—-both of them being not specific to neural networks—implementing and testing MENNAG was only a matter of hours. As a conclusion, we expect to be able to conduct many future work with a minimal implementation effort and a rigorous framework.

Nevertheless, this work brings to light some limitations of the attribute grammar paradigm for neural network encoding which should be pointed out. First, tree-based genotypes are easily expressed but many other genotypes could probably be used and could interact less nicely with attribute grammars. For instance, NEAT is very hard to express in this formalism due to its dependency to a niching algorithm and its custom cross-over operator. Another significant drawback is the computational cost of the attribute evaluation process. Noticeably, it can substantially slow down the evolutionary process when big sets are copied at each node of the trees. Nonetheless, quickly evaluating an attribute grammar is a well studied process and it should be possible to implement faster systems than our prototypes.

7 Future work

An important point of this work concerns its possible generalization. Thanks to the formalization, it is easy to evolve other structures than neural networks. To use MENNAG on other weighted directed graphs, one only has to change the interpretation of the allNodes and allConns lists. Moreover, many variants, for instance to evolve undirected graphs, can be designed by slightly modifying the presented grammar. Such uses will be studied in future work as well as the test of MENNAG on different neural-network problems.

The main technical issue revealed by the obtained results is the difficulty of correctly connecting I/Os when the dimension of the problem rises. In the robotic arm problem, there exists $6! \times 3! = 4{,}320$ different ways to link these I/O to a same, potentially optimal, structure. If one degree of freedom is added, the solution has to be found among more than one million different combinations. In consequence the main difficulty soon appears to lie in a good I/O choice rather than in exploring efficient modular structures. The proposed encoding has been designed to explore structures, so it will not be efficient enough to handle control problems with a high number of I/O. As a result, although modularity, hierarchy and repetition can lower the complexity of the search in cases of a high number of I/Os, for instance by using the same sub-network several times, they are not sufficient to handle an arbitrary number of I/Os.

Two approaches seem conceivable to cope with this central scalability issue. It has been suggested by some authors [7, 10, 17, 38] that using a geometric substrate to localize neurons can reduce the difficulty of the search by spatially gathering I/O that should work together. By defining the localization of neurons, users add knowledge and, consequently, face a trade-off between user-constraints and exploration. If users have almost no a priori information about the structure of the solutions, this substrate approach could be very difficult to use. Another approach could be to use development to link network building to I/O creation. This would imply evolving both morphology and control, as in [57].

8 Conclusion

In this work, we have presented MENNAG, a new encoding scheme designed to evolve neural-networks while exploiting modularity, hierarchy and repetition. It has been formalized by using an attribute grammar whose mechanisms have been described in detail. Compared to previous work on modular encodings, such as ModNet or ModularNEAT, MENNAG adds essentially hierarchy and formalization.

The performance of the new encoding has been compared to a direct encoding, NEAT, ModNet and a multilayer perceptron in two simple control problems. MENNAG led to better fitness than the other encodings in both cases. These encouraging results call for further studies of this new encoding, and further evaluations on new problems. Modules hierarchy and the way input/output connections are handled, for instance, ought to be more extensively tested.

Appendix A: Results of the Wilcoxon–Mann–Whitley test

See Tables 1, 2 and 3.

Table 1 Cartpole (two tails p values)

	Direct encoding	ModNet	MLP	MENNAG	NEAT
Direct encoding	1.000	0.077	0.171	0.011	0.237
ModNet	0.077	1.000	0.025	0.074	0.206
MLP	0.171	0.025	1.000	0.013	0.101
MENNAG	0.011	0.074	0.001	1.000	0.040
NEAT	0.237	0.206	0.101	0.040	1.000

Table 2 Robotic arm (two tails p values)

	MENNAG	ModNet	Direct encoding	NEAT	MLP
MENNAG	1.000	0.002	0.019	5×10^{-6}	0.395
ModNet	0.002	1.000	0.002	7×10^{-6}	4.6×10^{-6}
Direct	0.019	0.002	1.000	5×10^{-6}	3.07×10^{-6}
NEAT	5×10^{-6}	7×10^{-6}	5×10^{-6}	1.000	5×10^{-6}
MLP	0.395	5×10^{-6}	3×10^{-6}	5×10^{-6}	1.000

Table 3 Lesion experiments

	No clone	No cross-over	MENNAG (std)	No insert. mut.	Random mut.
No clone	1.000	0.712	7×10^{-5}	0.315	2×10^{-5}
No cross-over	0.719	1.000	5×10^{-5}	0.130	3×10^{-5}
MENNAG (std)	7×10^{-5}	6×10^{-5}	1.000	10×10^{-5}	0.898
No insert. mut.	0.315	0.130	10×10^{-5}	1.000	3×10^{-5}
Random mut.	2×10^{-5}	3×10^{-5}	0.898	3×10^{-5}	1.000

Appendix B: Complete attribute grammar

```
    // allN is the list of nodes
  // allC the list of connections
  // alll the list of input connections
  // allO the list of output connections
  [1.0,i1=0.5] START ⇒ DIV CONNS
      string DIV.id = ""
      set START.allN = DIV.allN
      string CONNS.divId = DIV.id
      set START.allC = union(set=DIV.allC, set=CONNS.allC)
      set START.alll = DIV.alll
      set START.allO = DIV.allO
  // n being the depth of the generated tree, this probability decreases when the depth increases.
  [1/(n*n)] DIV ⇒ DIV DIV CONNS
      string CONNS.divId = DIV.id
      string DIV_1.id = concatenate(DIV_0.id, "0")
      string DIV_2.id = concatenate(DIV_0.id, "1")
      set DIV.allC = union(set=DIV_1.allC, set=DIV_2.allC, set=CONNS.allC)
      set DIV.allN = union(set=DIV_1.allN, set=DIV_2.allN)
      set DIV.alll = union(set=DIV_1.alll, set=DIV_2.alll)
      set DIV.allO = union(set=DIV_1.allO, set=DIV_2.allO)
  //Similarly, this probability increases when the depth of the generated tree increases.
  [1.0-1.0/(n*n)] DIV ⇒ CELL CELL CONNS
      string CONNS.divId = DIV.id
      string CELL_0.divId = DIV.id
      string CELL_1.divId = DIV.id
      string CELL_0.id = concatenate(DIV.id, "0")
      string CELL_1.id = concatenate(DIV.id, "1")
      set DIV.allN = union(set=CELL_0.node, set=CELL_1.node)
      set DIV.allC = CONNS.allC
      set DIV.alll = union(set=CELL_0.alll, set=CELL_1.alll)
      set DIV.allO = union(set=CELL_0.allO, set=CELL_1.allO)
  // Implementation of the creation of neuron. The spec could be changed to create another kind of neuron. A
  potential input and output connections are associated to each neuron (cell).
  [1.0] CELL ⇒ IN OUT
      int CELL.spec = 0
      set CELL.node = createset(createhash(name="id", value=CELL.id, name="spec", value=CELL.spec))
      set CELL.alll = IN.alll
      set CELL.allO = OUT.allO
      string IN.divId = CELL.divId
      string OUT.divId = CELL.divId
      string IN.node = CELL.id
      string OUT.node = CELL.id
  // i1=probability to select this rule as r_1
  // i2=probability to select this rule as r_2
  [0.01,i2=0.5,i1=0.5] DIV ⇒ DIV CLONES CONNS
      string DIV_1.id = concatenate(DIV_0.id, "0")
      string CLONES.id = concatenate(DIV_0.id, "1")
      string CONNS.divId = DIV.id
      set CLONES.divN = DIV_1.allN
      set CLONES.divC = DIV_1.allC
      set CLONES.divI = DIV_1.alll
      set CLONES.divO = DIV_1.allO
      set DIV.tmp = addattribute(data=CLONES.allN, attname="cloneId", value=DIV_1.id)
      set DIV.allN = union(set=DIV_1.allN, set=DIV.tmp)
      set DIV.allC = union(set=CONNS.allC, set=DIV_1.allC, set=CLONES.allC)
      set DIV.alll = union(set=DIV_1.alll, set=CLONES.alll)
      set DIV.allO = union(set=DIV_1.allO, set=CLONES.allO)
  [0.01,i2=0.5,i1=1.5] DIV ⇒ CLONES DIV CONNS
      string DIV_1.id = concatenate(DIV_0.id, "1")
      string CLONES.id = concatenate(DIV_0.id, "0")
      string CONNS.divId = DIV.id
      set CLONES.divN = DIV_1.allN
      set CLONES.divC = DIV_1.allC
      set CLONES.divI = DIV_1.alll
      set CLONES.divO = DIV_1.allO
      set DIV.tmp = addattribute(data=CLONES.allN, attname="cloneId", value=DIV_1.id)
      set DIV.allN = union(set=DIV_1.allN, set=DIV.tmp)
      set DIV.allC = union(set=CONNS.allC, set=DIV_1.allC, set=CLONES.allC)
      set DIV.alll = union(set=DIV_1.alll, set=CLONES.alll)
      set DIV.allO = union(set=DIV_1.allO, set=CLONES.allO)
  // list of clones
  [0.01,i2=0.5] CLONES ⇒ CLONE CLONES
      string CLONE.id = concatenate(CLONES_0.id, "0")
      string CLONES_1.id = concatenate(CLONES_0.id, "1")
      set CLONE.divN = CLONES.divN
      set CLONE.divC = CLONES.divC
      set CLONE.divI = CLONES.divI
      set CLONE.divO = CLONES.divO
      set CLONES_1.divN = CLONES.divN
      set CLONES_1.divC = CLONES.divC
      set CLONES_1.divI = CLONES.divI
      set CLONES_1.divO = CLONES.divO
      set CLONES.allN = union(set=CLONE.allN, set=CLONES_1.allN)
      set CLONES.allC = union(set=CLONE.allC, set=CLONES_1.allC)
      set CLONES.alll = union(set=CLONE.alll, set=CLONES_1.alll)
      set CLONES.allO = union(set=CLONE.allO, set=CLONES_1.allO)
  // End of the list of clones
  [1.0] CLONES ⇒ CLONE
      string CLONE.id = CLONES.id
      set CLONE.divN = CLONES.divN
      set CLONE.divC = CLONES.divC
      set CLONE.divI = CLONES.divI
      set CLONE.divO = CLONES.divO
      set CLONES.allN = CLONE.allN
      set CLONES.allC = CLONE.allC
      set CLONES.alll = CLONE.alll
      set CLONES.allO = CLONE.allO
  // Implementation of the clone
  // permi is the vector of permutation for inputs, permo the one for outputs. 2 is the number of inputs and 1 the
  number of outputs. These numbers have to changed if you add new I/Os
  // The CLONE.id is concatenated to each id found in the cloned DIV. I/Os are permutated using permo and permi.
  [1.0] CLONE ⇒ end
      evofloat[2] CLONE.permi = evolved
      evofloat[1] CLONE.permo = evolved
      set CLONE.allN = prependtoattributes(data=CLONE.divN, attname="id", value=CLONE.id)
      set CLONE.tmp = prependtoattributes(data=CLONE.divC, attname="target", value=CLONE.id)
      set CLONE.allC = prependtoattributes(data=CLONE.tmp, attname="node", value=CLONE.id)
      set CLONE.tmpii = prependtoattributes(data=CLONE.divI, attname="node", value=CLONE.id)
      set CLONE.tmpi = prependtoattributes(data=CLONE.tmpii, attname="target", value=CLONE.id)
      set CLONE.alll = map(data=CLONE.tmpi, pushedname="data",
  foncteur=setattribute(value=aspermutation(data=CLONE.permi, index=getattribute(data=⇒data, attname="spec")),
  attname="spec"))
      set CLONE.tmpoo = prependtoattributes(data=CLONE.divO, attname="target", value=CLONE.id)
      set CLONE.tmpo = prependtoattributes(data=CLONE.tmpoo, attname="node", value=CLONE.id)
      set CLONE.allO = map(data=CLONE.tmpo, pushedname="data",
  foncteur=setattribute(value=aspermutation(data=CLONE.permo, index=getattribute(data=⇒data, attname="spec")),
  attname="spec"))
  [0.5] CONNS ⇒ CONN CONNS
      string CONN.divId = CONNS.divId
      string CONNS_1.divId = CONNS.divId
      set CONNS.allC = union(set=CONNS_1.allC, set=CONN.allC)
  [0.5] CONNS ⇒ end
      set CONNS.allC = createset()
  // This variant of CONN creates a feed-forward connection by forcing the source of the connection to be in the left
  branch of the tree and the target to be in right branch. This is implemented by modifying the Ids.
  [0.9] CONN ⇒ ICONN
      set CONN.allC = ICONN.allC
      string ICONN.targetBaseId = concatenate(CONN.divId, "1")
      string ICONN.nodeBaseId = concatenate(CONN.divId, "0")
  // This is the basic (not feed-forward) connection
  [0.1] CONN ⇒ ICONN
      set CONN.allC = ICONN.allC
      string ICONN.targetBaseId = CONN.divId
      string ICONN.nodeBaseId = CONN.divId
  // Implementation of a connection.
  // A connection is made of a source node, a target, a weight and a specification (spec). The spec could be changed
  to create different kind of connections.
  [1.0] ICONN ⇒ weight node target
      evofloat weight.value = evolved
      bitstring target.value = evolved
      bitstring node.value = evolved
      float weight.float = numericalvalue(weight.value)
      string ICONN.target = concatenate(ICONN.targetBaseId, target.value)
      string ICONN.node = concatenate(ICONN.nodeBaseId, node.value)
      set ICONN.allC = createset(createhash(name="node", value=ICONN.node, name="target",
  value=ICONN.target, name="weight", value=weight.float, name="spec", value=0, name="type", value="internal"))
  // An empty IN connection (no connection to sensors for this neuron)
  [0.5] IN ⇒ end
      set IN.alll = createset()
  // Connection to the first sensor.
  [0.3,p2=0.1] IN ⇒ IO in1
      int IO.spec = 0
      set IN.alll = IO.allC
      string IO.divId = IN.divId
      string IO.node = IN.node
      string IO.type = "in"
  // Connection to the second sensor.
  [0.0,p2=0.1] IN ⇒ IO in2
      int IO.spec = 1
      set IN.alll = IO.allC
      string IO.divId = IN.divId
      string IO.node = IN.node
      string IO.type = "in"
  [0.5] OUT ⇒ end
      set OUT.allO = createset()
  // Connection to the first actuator
  [0.3,p2=0.1] OUT ⇒ IO out1
      int IO.spec = 0
      set OUT.allO = IO.allC
      string IO.divId = OUT.divId
      string IO.node = OUT.node
      string IO.type = "out"
  // Add here new in and out if needed
  // Implementation of IO connections
  // The source node is the node passed down when the cell has been created;
  [1.0] IO ⇒ weight target
      evofloat weight.value = evolved
      bitstring target.value = evolved
      string IO.target = IO.node
      float weight.float = numericalvalue(weight.value)
      set IO.allC = createset(createhash(name="node", value=IO.node, name="target", value=IO.target,
  name="weight", value=weight.float, name="spec", value=IO.spec, name="type", value=IO.type))
```

Appendix C: Defined types and functions

Mennag uses a short list of types and functions to handle attributes. Here is a short reference.

Seven basic types are used in the MENNAG grammar:

- string: character string
- bitstring: string of bits which can mutate (bit flipping)
- float: real number
- evofloat: real number which can mutate (Gaussian mutation)
- int: integer
- set: a simple set which can handle any type
- hash: a simple hash table

A vector of these types can be defined by adding the size enclosed in brackets. For instance, "evofloat[10]" defines a vector of 10 evofloat.

Hash tables are used to emulate a structured type (struct in C). For instance, a neuron is "created" using "createhash(name = "id", value = CELL.id, name = "spec", value = CELL.spec)".

Some simple functions are built-in:

- union: merge two sets
- concatenate: concatenate two strings
- createset: create an empty set
- createhash: create an empty hash table
- numericalvalue: convert the argument to a real number
- prependtoattributes: concatenate a string in front of all the specified entries in all the hash tables contained in a set
- addattribute: add an entry to every hash tables contained in a set
- map: apply a functor to every elements of a set
- aspermutation: compute the permutation corresponding to a vector a real numbers

Appendix D: Experimental setup

D.1 MENNAG

- cross rate: 0.3
- min. number of neurons (random generation): 2
- max. number of neurons (random generation): 10
- min. number of connections (random generation): 2
- min. number of connections (random generation): 12
- mutation rate (random tree): 0.05
- insertion mutation rate: 0.2
- mutation delete rate: 0.1

D.2 NEAT

- starting network (cart-pole): each input directly connected to the output
- parameter file: p2nv.ne (available in NEAT's source code)

D.3 ModNet

- max. number of modules: 10
- cross rate: 0.6
- no predefined module in the initial module pool
- add model module rate: 0.01
- insert module rate: 0.01
- delete model module rate: 0.005
- mutation module rate: 0.05

D.4 Direct encoding

- max. number of neurons: 12
- min. number of neurons: 2
- max. number of connections: 10
- min. number of connections: 2
- add neuron rate: 0.1
- delete neuron rate: 0.1
- add connection rate: 0.25
- delete connection rate: 0.25
- change connection rate: 0.25

References

1. Anderson JR (2007) How can the human mind occur in the physical Universe? In: The modular organization of the mind. Oxford University Press, New York, pp 45–91
2. de Beeck HPO, Haushofer J, Kanwisher NG, et al (2008) Interpreting fMRI data: maps, modules and dimensions. Nat Rev Neurosci 9:123–135
3. Boers JWE, Kuiper H, Happel BLM, Springhuizen-Kuiper IG (1993) Designing modular artificial neural networks. Technical report, Rijksuniversiteit te Leiden
4. Bowers CP, Bullinaria JA (2005) Embryological modelling of the evolution of neural architecture. In: A Cangelosi GB, Borisyuk R (eds) Modeling language, cognition and action. World Scientific, Singapore, pp 375–384
5. Buessler JL, Urban JP (2002) Biologically inspired robot behavior engineering. In: Modular neural architectures for robotics. Springer, Heidelberg
6. Bullinaria JA (2007) Understanding the emergence of modularity in neural systems. Cogn Sci 31(4):673–695
7. Cangelosi A, Parisi D, Nolfi S (1994) Cell division and migration in a 'genotype' for neural networks. Netw Comput Neural Syst 5(4):497–515

8. Cantu-Paz E, Kamath C (2005) An empirical comparison of combinations of evolutionary algorithms and neural networks for classification problems. IEEE Trans Syst Man Cybern Part B 35(5):915–927
9. Cliff D, Harvey I, Husbands P (1992) Incremental evolution of neural network architectures for adaptive behaviour. Technical report. Cognitive science research paper CSRP256, Brighton BN1 9QH, England, UK
10. D'Ambrosio D, Stanley K (2007) A novel generative encoding for exploiting neural network sensor and output geometry. In: Proceedings of the 9th annual conference on genetic and evolutionary computation, pp 974–981
11. Deb K (2001) Multi-objectives optimization using evolutionnary algorithms. Wiley, New York
12. Deb K, Goldberg DE (1989) An investigation of niche and species formation in genetic function optimization. In: Proceedings of the third international conference on genetic algorithms. Morgan Kaufmann Publishers, San Francisco, pp 42–50
13. Doncieux S (2003) Évolution de contrôleurs neuronaux pour animats volants : méthodologie et applications. PhD thesis, LIP6/AnimatLab, Université Pierre et Marie Curie, Paris, France
14. Doncieux S, Meyer JA (2004a) Evolution of neurocontrollers for complex systems: alternatives to the incremental approach. In: Proceedings of the international conference on artificial intelligence and applications (AIA 2004)
15. Doncieux S, Meyer JA (2004b) Evolving modular neural networks to solve challenging control problems. In: Proceedings of the fourth international ICSC symposium on engineering of intelligent systems (EIS 2004)
16. Doncieux S, Meyer JA (2005) Evolving PID-like neurocontrollers for non-linear control problems. Int J Control Intell Syst (IJCIS) Spec Issue Nonlinear Adapt PID Control 33(1):55–62
17. Fukutani I, Vaario J (1997) The effect of environment to genetic growth. In: International symposium on system life, July 21–22, 1997, Tokyo, Japan, pp 227–232
18. Gallinari P (1998) Modular neural net systems, training of the handbook of brain theory and neural networks table of contents, pp 582–585
19. García-Arnau M, Manrique D, Ríos J, Rodríguez-Patón A (2007) Initialization method for grammar-guided genetic programming. Knowl Based Syst 20(2):127–133
20. Gauci J, Stanley K (2007) Generating large-scale neural networks through discovering geometric regularities. In: Proceedings of the 9th annual conference on Genetic and evolutionary computation, pp 997–1004
21. Geva S, Sitte J (1993) A cartpole experiment benchmark for trainable controllers. Control Systems Mag IEEE 13(5):40–51
22. Gomez FJ, Schmidhuber J, Miikkulainen R (2006) Efficient non-linear control through neuroevolution. In: Proceedings of the European conference on machine learning (ECML-06, Berlin), pp 654–662
23. Grönroos M (1999) A comparison of some methods for evolving neural networks. In: Proceedings of the genetic and evolutionary computation conference, vol 2. Morgan Kaufmann, Menlo Park, p 1442
24. Gruau F (1995) Automatic definition of modular neural networks. Adapt Behav 3(2):151–183
25. Gruau F, Whitley D, Pyeatt L (1996) A comparison between cellular encoding and direct encoding for genetic neural networks. In: John Koza R, David Goldberg E, David Fogel B, Rick Riolo L (eds) Genetic programming 1996: proceedings of the first annual conference. MIT Press, Stanford University, CA, pp 81–89
26. Hartwell L, Hopfield J, Leibler S, Murray A (1999) From molecular to modular cell biology. Nature 402(6761):C47–C52
27. Holland JH (1975) Adaptation in natural and artificial systems. University of Michigan Press, MI
28. Hornby G (2005) Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In: Proceedings of the 2005 conference on genetic and evolutionary computation. ACM Press, New York, pp 1729–1736
29. Hornby G, Pollack J (2002) Creating high-level components with a generative representation for body–brain evolution. Artif Life 8(3):223–246
30. Huettel SA, Song AW, McCarthy G (2004) Functional magnetic resonance imaging. Sinauer Associates, Sunderland
31. Hussain T (2003) Attribute grammar encoding of the structure and behaviour of artificial neural networks. PhD thesis, Queen's University
32. Igel C (2003) Neuroevolution for reinforcement learning using evolution strategies. In: The 2003 congress on evolutionary computation, CEC'03, vol 4. IEEE Press, Ne wYork, pp 2588–2595
33. de Jong E, Thierens D (2004) Exploiting modularity, hierarchy, and repetition in variable-length problems. In: Proceedings of the genetic and evolutionary computation conference, GECCO-04. Springer, Heidelberg, pp 1030–1041
34. Kashtan N, Alon U (2005) Spontaneous evolution of modularity and network motifs. Proc Natl Acad Sci 102(39):13,773–13,778
35. Kitano H (1990) Designing neural networks using genetic algorithms with graph generation system. Complex Syst 4:461–476
36. Knuth D (1968) Semantics of context-free languages. Theory Comput Syst 2(2):127–145
37. Kodjabachian J, Meyer JA (1995) Evolution and development of control architectures in animats. Robot Auton Syst 16:161–182
38. Kodjabachian J, Meyer JA (1997) Evolution and development of neural networks controlling locomotion, gradient-following, and obstacle-avoidance in artificial insects. IEEE Trans Neural Netw 9:796–812
39. Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge
40. Landau S, Doncieux S, Drogoul A, Meyer JA (2002) SFERES: un framework pour la conception de systèmes multi-agents adaptatifs. Technique et Science Informatiques 21(4):427–446
41. Lindenmayer A (1968) Mathematical models for cellular interaction in development, parts i and ii. Journal of theoretical biology 18(18):280–315
42. Lipson H (2004) Principles of Modularity, Regularity, and Hierarchy for Scalable Systems. In: Genetic and evolutionary computation conference (GECCO'04) workshop on modularity, regularity and hierarchy
43. Lipson H, Pollack JB, Suh NP (2002) On the origin of modular variation. Evolution 56(8):1549–1556
44. Luke S, Spector L (1996) Evolving graphs and networks with edge encoding: preliminary report. In: Late breaking papers at the genetic programming 1996 conference, pp 117–124
45. Mattiussi C, Floreano D (2004) Evolution of analog networks using local string alignment on highly reorganizable genomes. In: Evolvable hardware, 2004. Proceedings of conference on NASA/DoD 2004, pp 30–37
46. Mattiussi C, Floreano D (2007) Analog genetic encoding for the evolution of circuits and networks. IEEE Trans Evol Comput 11:596–607
47. Michel O, Clergue M, Collard P (1997) Artificial neurogenesis: Applications to the cart-pole problem and to an autonomous mobile robot. International Journal on Artificial Intelligence Tools 6(4):613–634
48. Miller GF, Todd PM, Hedge SU (1989) Designing neural networks using genetic algorithms. In: Proceedings of the third international conference on artificial intelligence. Morgan Kaufmann, Menlo Park, pp 762–767
49. Mouret JB, Doncieux S, Meyer JA (2006) Incremental evolution of target-following neuro-controllers for flapping-wing animats.

In: Nolfi S, Baldassare G, Calabretta R, Hallam J, Marocco D, Meyer JA, Miglino O, Parisi D (eds) From animals to animats: proceedings of the 9th international conference on the simulation of adaptive behavior (SAB), Rome, Italy, pp 606–618
50. Nolfi S, Parisi D (1998) "Genotypes" for neural networks. The handbook of brain theory and neural networks, pp 431–434
51. Paakki J (1995) Attribute grammar paradigms: a high-level methodology in language implementation. ACM Comput Surv (CSUR) 27(2):196–255
52. Pasemann F, Dieckmann U (1997) Evolved neurocontrollers for pole-balancing. Biol Artif Comput Neurosci Technol Proc IWANN 97:1279–1287
53. Reisinger J, Miikkulainen R (2007) Acquiring evolvability through adaptive representations. In: Proceedings of the 9th annual conference on Genetic and evolutionary computation. ACM Press, New York, pp 1045–1052
54. Reisinger J, Stanley K, Miikkulainen R (2004) Evolving reusable neural modules. In: Proceedings of the genetic and evolutionary computation conference (GECCO-2004). Springer, Heidelberg, pp 69–81
55. Siddiqi A (1998) Comparison of matrix rewriting versus direct encoding for evolving neural networks. In: The 1998 IEEE international conference on evolutionary computation, ICEC'98, pp 392–397
56. Simon H (1962) The architecture of complexity. Proc Am Philos Soc 106(6):467–482
57. Sims K (1994) Evolving 3d morphology and behavior by competition. In: Brooks RA, Maes P (eds) Proceedings of the fourth international workshop on artificial life. The MIT Press, Cambridge, pp 28–39
58. Stanley K, Miikkulainen R (2002) Evolving neural networks through augmenting topologies. Evol Comput 10(2):99–127
59. Stanley KO (2006) Comparing artificial phenotypes with natural biological patterns. In: Proceedings of the genetic and evolutionary computation conference (GECCO) workshop program, New York, NY
60. Teuber HL (1955) Physiological psychology. Annu Rev Psychol 6(1):267–296
61. Wagner G, Altenberg L (1996) Complex adaptations and the evolution of evolvability. Evolution 50(3):967–976
62. Watson R (2005) Modular interdependency in complex dynamical systems. Artif Life 11(4):445–457
63. Widrow B (1987) The original adaptive neural net broom-balancer. In: International symposium on circuits and systems, pp 351–357
64. Wieland A (1991) Evolving neural network controllers for unstable systems. In: Proceedings of the international joint conference on neural networks (Seattle, WA). IEEE, Piscataway, pp 667–673
65. Yao X (1999) Evolving artificial neural networks. Proc IEEE 87(9):1423–1447