An Investigation Into the Structure of Genomes within an Evolution that uses Embryogenesis

Anthony M. Roy
Engineering Design Research Laboratory
California Institute of Technology
1200 California Blvd.
Pasadena CA 91125
roy@design.caltech.edu

Erik K. Antonsson
Engineering Design Research Laboratory
California Institute of Technology
1200 California Blvd.
Pasadena CA 91125
erik@design.caltech.edu

Andrew A. Shapiro
Engineering Design Research Laboratory
California Institute of Technology
1200 California Blvd.
Pasadena CA 91125
aashapiro@aol.com

ABSTRACT

Evolutionary algorithms that use embryogenesis in the creation of individuals have several desirable qualities. Such algorithms are able to create complex, modular designs which can scale well to large problems. However, the inner workings of developmental algorithms have not been investigated as thoroughly as their direct-encoding counterparts. More precisely, it would be beneficial to look at how the rules used during embryogenesis evolve alongside the phenotypes they produced. This paper reports on such an investigation into the evolution of a rule set for the growth of an artificial neural network, and identifies several aspects that are desirable for the genomes of a developmental evolutionary algorithm.

Categories and Subject Descriptors

I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*Analysis of algorithms*; I.2.6 [**Artificial Intelligence**]: Learning—*connectionism and neural nets*

General Terms

Algorithms

Keywords

Neural Networks, Embryogenesis, Analysis

1. INTRODUCTION

Encoding schemes that apply embryogenesis are emerging as a new way to encode genotypes for Genetic Algorithms (GA's). Theraulaz [11] and Pollack [8] have shown that the reuse of a small set of rules to create a phenotype is an effective alternative to storing and manipulating the large amount of data that describes each individual directly. Bentley and Kumar [1] have shown that indirect encodings produce solutions to design problems faster and better than their directly encoded counterparts. Federici and Downing [2] have also shown that rule-based encoded designs are more robust as well. Furthermore, these encodings are much more likely to create modular outcomes, another desirable quality for many designs [4, 6, 3].

Considering these many advantages, indirect encodings have been used for many design problems, among them creating artificial neural networks (ANN's)[5, 10]. The complexity and size of ANN's make them attractive candidates for Evolutionary Computation (EC) applications.

This paper explores the inner workings of an implicitly encoded genetic algorithm. The genome is a set of variable-length rules that are decoded to create a C^{++} program. The C^{++} programs used to create the ANN's have an *IF-CONDITION, THEN-ACTION* structure. Each program cycles through each node with multiple tests and actions of the form:

IF Node α and/or Node β meets certain CONDITION(S), THEN perform ACTION(S).

As with EC itself, the work presented here in inspired by nature. Rather than using a tree-like structure as in most ANN embryogenesis, the genome is represented as a series of numbers ranging between 1 and 4. This representation allows one to evolve with mutations such as point substitution, crossover, and gene duplication/deletion in a manner much closer to their biological counterparts [7]. The genome is then created to create a C^{++} program with a process that is akin to Grammatical Evolution (GE) [12]. A key difference, however, is that the programs created by GE are the ANN's, while the programs produced here are instructions on how to make the ANN. While this difference is subtle, the recursive execution of rules leads to inherent modularity [9].

This paper will investigate how the rules evolve with their accompanying individuals. In it, we see that increasing rule complexity is integral to a successful, rule-based evolution. Furthermore, we see that events of punctuated equilibrium are a key feature of successful evolution. In evolutionary biology, punctuated equilibrium is an event where long periods where a species changes little are "punctuated" by relatively short periods of dramatic changes in the individual. While the term is normally applied on a phenotypical level, we will see that it also applies to the genomes as well.

2. PHENOTYPES

The goal of this evolutionary algorithm is to evolve a neural network capable of effectively controlling a robot, even as the network has several nodes and connections removed. This test was chosen because it challenges the exploration capabilities of the GA in finding a suitable controller, and

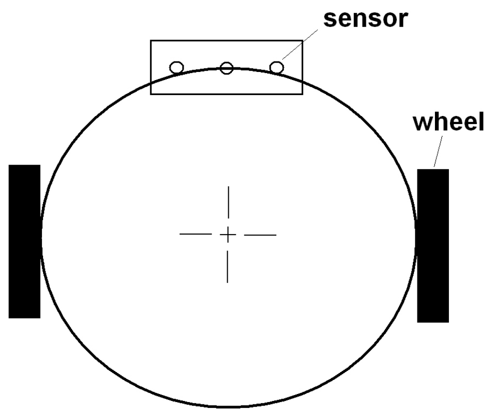


Figure 1: Model of Path-Following Robot

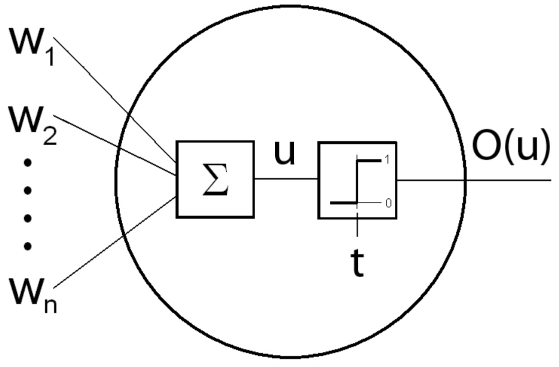


Figure 2: McCulloch-Pitts Neuron Model

$$\underbrace{\underbrace{\underbrace{11}_{\text{codon}}\ 2341233123}_{\text{reading frame}}\ 141434132124\ 444444444444}_{\text{gene}}$$

Figure 3: Sample Genome and Biological Analog

also challenges the exploitative capabilities of the GA in improving the robustness of the controller. The robot considered here is a Pioneer robot equipped with photo-voltaic sensors. These sensors are able to detect the difference between light and dark, and are configured to enable the robot to follow a desired path via a line.

The neural network controller of each robot is a network of McCulloch-Pitts neurons. The sensors serve as the inputs to the ANN, while each output of the ANN controls a wheel. Each ANN is composed of McCulloch-Pitts modeled neurons shown in Figure 2. The neuron sums the weighted inputs, then enters the sum into a Heaviside function with an evolvable threshold.

The ANN is updated at the same time as the robot's position, so large ANN's can experience noticeable lag times. The node's output, O(u), maybe weighted before it is used as an input for another node. However, O(u) for an output node is always unweighted, resulting in binary outputs for the entire ANN. Finally, each node is denoted by two different types. Type 1 values indicate whether a node is an INPUT, HIDDEN, or OUTPUT. Type 2 values are numerical values that range between 1 and 8, and nodes are enumerated in the order in which they are created. Thus, the third hidden node created by the ANN would have a *type 1* of HIDDEN and a *type 2* of 3.

3. GENETIC ENCODING

3.1 Biological Analog

It will be helpful to review the biological analogy which was the inspiration for this particular encoding scheme. The genome of each individual is an array of integers which is decoded to create C^{++} programs. Similar to the quaternary system of natural genetics, every digit is a *nucleotide* whose value is inclusively between 1 and 4. It takes a pair of nucleotides to write anything into the C^{++} script. As shown in Figure 3, two nucleotides are analogous to a *codon* transcribing an amino acid into a protein. A collection of six codons (twelve nucleotides) forms a complete if/then statement, which can be thought of as the secondary structure of a protein. However, these tests are not independent, and the sequence of the tests will greatly influence how the individual will grow. Therefore, a combination of these tests determines what actions will be performed, and can be thought of as the overall protein. Herein, a *reading frame* is a collection of six codons and the sequence of frames that creates an entire protein as a *gene*.

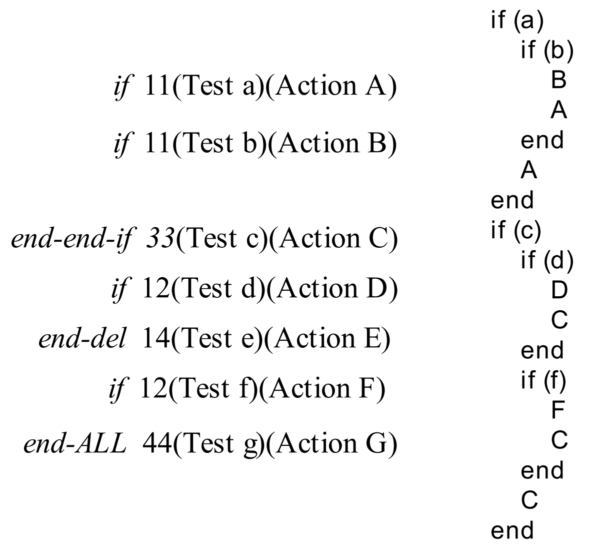
3.2 Genotypes and Genetic Decoding

The genome is a set of variable-length rules that are decoded to create a program. While a brief overview will be given here, a more details on the particulars of how the genome is decoded can be found in [9]. The programs have an IF/THEN structure, where criteria are tested, and if true, certain actions are performed. The first codon of each reading frame dictates how the criteria and actions are grouped. This flexibly allows the GA to build complex rules from simple building blocks. The first pair of nucleotides in a frame determines the logical structure of the entire protein. An example is shown in Figure 4.

- *if* - Opens an if statement. Adds action to the action stack.
- *end-if* - Executes action stack. Removes most recent action from action stack. Closes an if statement. Opens another if statement. Adds action to the action stack.
- *end-end-if* - Executes action stack. Removes most recent action from action stack. Closes an if statement. Executes new action stack. Removes action from action stack. Closes an if statement. Opens another if statement. Adds action to the action stack.
- *end-del* - Executes action stack. Closes an if statement.
- *end-end-del* - Executes action stack. Removes most recent action from action stack. Closes an if statement.

Executes new action stack. Removes action from action stack. Closes an if statement.

- *end-ALL* - Executes action stack. Removes most recent action from action stack. Closes an if statement. Repeats until all if statements are closed.



```
if 11(Test a)(Action A)                 if (a)
                                           if (b)
if 11(Test b)(Action B)                       B
                                              A
                                           end
end-end-if 33(Test c)(Action C)            A
                                        end
if 12(Test d)(Action D)                 if (c)
                                           if (d)
end-del 14(Test e)(Action E)                  D
                                              C
if 12(Test f)(Action F)                    end
                                           if (f)
end-ALL 44(Test g)(Action G)                  F
                                              C
                                           end
                                           C
                                        end
```

Figure 4: If Structure Codon and Protein Transcription

The remaining codons of a reading frame determine the test to be performed and the actions that will be executed upon a successful test. The second codon dictates what attributes will be tested. The attributes are node states such as the type of node (OUTPUT-4, *etc.*) or what previous actions a node has performed. The third codon determines what (in)equality will be used for the test. The fourth pair of nucleotides assigns the values the attribute is tested against. Each possible value correlates to an integer in the range [1 - 16]. Table 1 shows an overview of the codons and their functions.

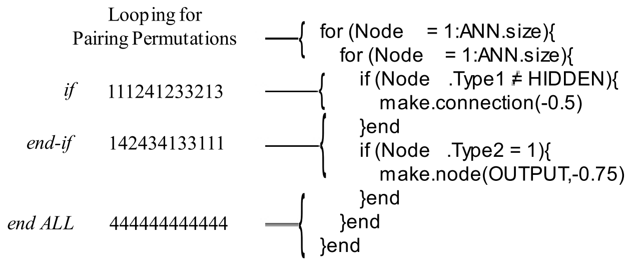
Table 1: List of codons

Codon	Function	Example
1	If Logic Structure	*if, END ALL*
2	Condition	*Node β.no_of_inputs()*
3	Test Inequality	$\neq$, $=$, $>$
4	Test Value	*OUTPUT, 3, -0.75*
5	Action	*Make Node, Make Connection*
6	Action Value	*OUTPUT, 3, -0.75*

The fifth codon determines what action will be placed into the action stack. This "stack" of actions is written into the program whenever an if statement is closed. Some codons will result in the creation of a new node. Others will create a connection between Node α and Node β. In both these cases, the last two nucleotides dictate the threshold of the new node or weight of the connection, respectively. The transcription options are identical to the bias and connection values of the fourth codon. Thus the frame *112341233123* will create:

```
if (Node $\alpha$.Type1() $\neq$ HIDDEN)
    make connection(0.5)
```

There are also the action options of *End Turn* and *Do Nothing* which will exit the permutation loop and not insert any action, respectively. Figure 5 shows the genetic string used to create a C^{++} program.



```
for (Node    = 1:ANN.size){
  for (Node    = 1:ANN.size){
    if (Node    .Type1 ≠ HIDDEN){
      make.connection(-0.5)
    }end
    if (Node   .Type2 = 1){
      make.node(OUTPUT,-0.75)
    }end
  }end
}end
```

Figure 5: Sample Genome and Protein Pseudo-code

3.3 C^{++} Programs (Proteins)

Each C^{++} program is a collection of proteins that build the phenotype. While the genome creates the bulk of the algorithm, there are a few rules hard coded into the C^{++} script of every individual. First, every ANN begins as three input nodes with a threshold of zero. As there is no option to create another input, each ANN will contain exactly three input nodes. Furthermore, the inputs are unable to connect to each other. Also, each input, and all subsequent nodes thereafter, can create up to one additional node of either a hidden layer type or output type.

As we are only considering feed-forward ANN's, nodes are only able to make connections to nodes created after them. Furthermore, the creation of an output node will stop the creation of any other nodes. However, connections may still be grown at this point. The act of creating a node or connection consumes one of the individual's predetermined energy units for the entire ANN. The individual is considered to be completely developed once the individual uses all energy units or the programs cycles through all pairing permutations of nodes without performing any actions. These hard coded rules are implemented to impose the minimum constraints any viable feed forward ANN would have, while leaving enough flexibility to create a variety of architectures. Figure 6 shows the development of a NAND gate using the pseudo-code from Figure 5. It is important to note that an infinite number of genomes could have created an identical ANN.

4. GENETIC ALGORITHM

Each evolutionary run begins with the random creation of 192 individuals, each having genome lengths of 600 nucleotides. After the embryogenesis of each individual, as described by the method above, each ANN is evaluated. The fitness function uses a modified tier system with an individual being rewarded exponentially for each goal. Equation 1

```
for (Node    = 1:ANN.size){
   for (Node    = 1:ANN.size){
      if (Node    .Type1≠ HIDDEN){
         make.connection(0.5)
      }end
      if (Node    .Type2 = 1){
         make.node(OUTPUT,-0.75)
      }end
   }end
}end
```

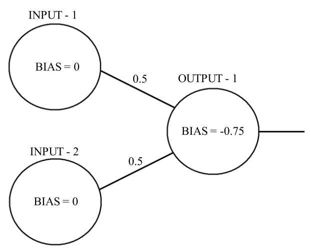


Figure 6: Protein Pseudo-code and Sample NAND Gate

is the fitness function used for evaluating individuals. The exponent in Equation 1 is decided by the tier system in Table 3. The first tier ensures the individual grows the correct number of output nodes. In the second tier, the exponent is increased for each output node with a connection. These two requirements are the minimum for any possibly viable ANN circuit, and once met, will yield an exponent of $x-1 = 3$. At this point, the network's truth table is compared with logic necessary to navigate a line. This logic is shown in Table 2. The exponent gets an additional point for each correct answer. Two of the logic combinations, [1 0 1] and [1 1 1], are not considered as there is no physical realization in the former and the robot has already failed if it encounters the latter. After tier 3, a completely functional robot will have $x - 1 = 9$ and an overall fitness of 512.

Table 2: Target Logic for Robot

Input			Output	
Left Sensor	Center Sensor	Right Sensor	Left Wheel	Right Wheel
0	0	0	1	1
0	0	1	1	0
0	1	0	1	1
0	1	1	1	0 or 1
1	0	0	0	1
1	1	0	0 or 1	1

In tier 4, a connection is randomly broken, and the ANN is compared to the target logic again. Connections are continually broken until the circuit no longer produces the target logic. Once the circuit fails, the connections are replaced and the process is repeated with broken nodes. This test for robustness is performed once for each generation the individual is alive. Because the order in which the connections and nodes are removed changes each generation, an individual's fitness is not constant, and the overall robustness will increase.

$$Fitness = \lfloor 2^{x-1} \rfloor \tag{1}$$

A roulette style of selection determines which individuals are used for creating the next generation, and population size is conserved. The probability of selecting an individual is its fitness value divided by the summed fitness of the entire population. 25% of the population of the current generation live on to the next generation. The remaining 75% is created through the various mutation methods where each individual can have up to three offspring. The mutations between the parent and offspring are point mutation and double point crossover. These parameters were chosen to give reasonable computational results within 1000 generations.

Table 3: Tier for adjusting fitness exponent $(x - 1)$

Tier	Test	Change in Exponent
1	Are there enough output nodes?	# of desired output nodes
2	Are there a connections to each output node?	+ # of output nodes with connections
3	Compare to the desired truth table	+ # of correct answers in each table entry
4	Break connections and nodes until failure	+ % of connections broken + % of nodes broken

5. EVOLUTIONARY RESULTS

Evolution was run with two different mutation rates. For one group of runs, each reading frame has a 10% chance of undergoing a mutation. For a second group, the mutation rate was increased to 80%. The lower mutation rate yielded good results, as individuals were able to follow the line with large portions of their networks removed. Evolution using the higher mutation rate were deemed unsuccessful, as individuals seldom made it to the 4th evaluation tier. Evolution was run several times for each scenario, but in the interest of space only the results for two exemplary runs are shown below. While the quantitative results differ between runs, the qualitative results for all the runs are similar to these runs.

Figure 7 shows what genes were used by the best individuals (top 10%) throughout a typical successful evolution. Each time a gene is used, a dot is placed that shows in which generation it was used. Furthermore, the figure is overlaid with a plot of the fitness of the best performing individual of each generation. Punctuated equilibrium (PE) events happen near generations 270 and 610. The first PE event happens shortly after the first jump in fitness of the best individual. The second PE event happens after a relatively small change (∼1%) increase in the best fitness. Finally, we see that the majority of increases in best fitness do not result in a massive shift of the genomes in the population.

The analysis was repeated for poorly-performing evolutions with the elevated mutation rate. On a phenotypical level, no individual ever made it to the 4th tier. Figure 8 shows the effects a high mutation rate has on genome development. The most striking feature is the lack of any PE's. Another key element is the myriad genes created as the number of genes used among the top 10% here is an order of magnitude higher than the number genes used in a successful evolution.

Figure 9 shows how the rules become more complex throughout evolution. The height of the overall bar diagram shows how many different genes were used throughout evolution grouped by generation groups of one hundred. The number of nestings indicate the number of additional conditions that must prove true in order for an action to be executed. Thus, a thrice nested rule must have four IF statements true

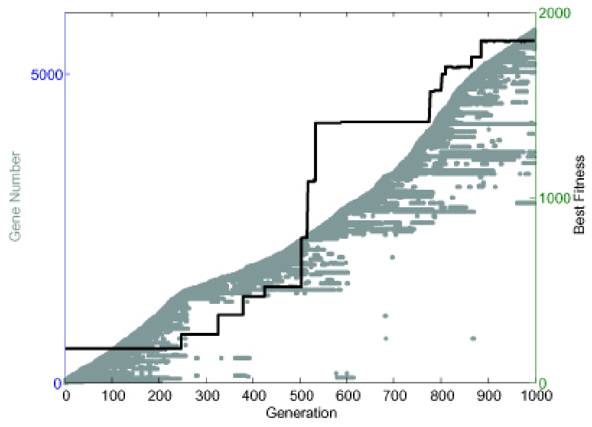


Figure 7: Lifetime Genes Used by the Top 10% of Each Generation During a Successful Evolution

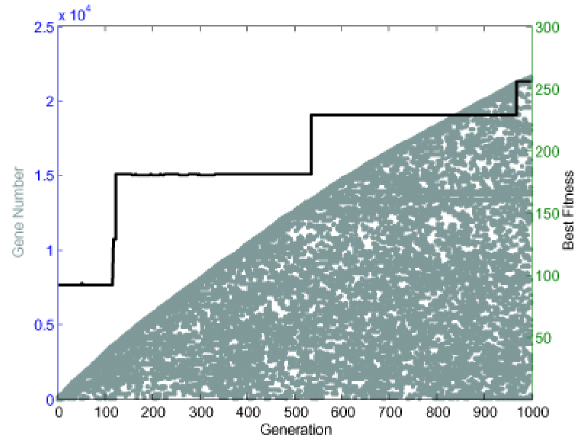


Figure 8: Lifetime Genes Used by the Top 10% of Each Generation During a Unsuccessful Evolution

for its action to execute. We see that over time, a higher percentage of the rules used have additional nestings. Furthermore, the number of genes used by the best individuals changes as well. Unsuccessful results are also contrasted in Figure 10. Here we see, once again, that many more genes are produced. However, there is little variation throughout evolution. Furthermore, the rules used do not become overly complex.

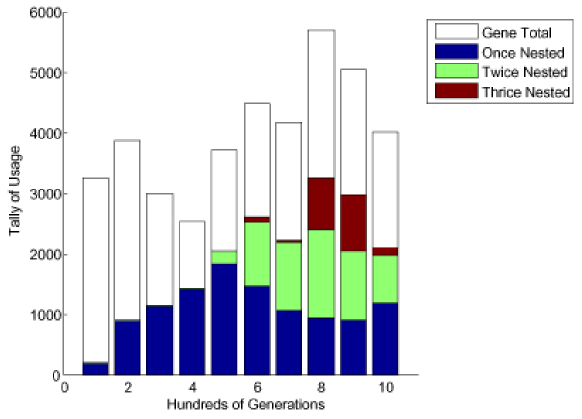


Figure 9: Structure of Genes Used by the Top 10% of Each Generation During a Successful Evolution (Once Nested at Bottom)

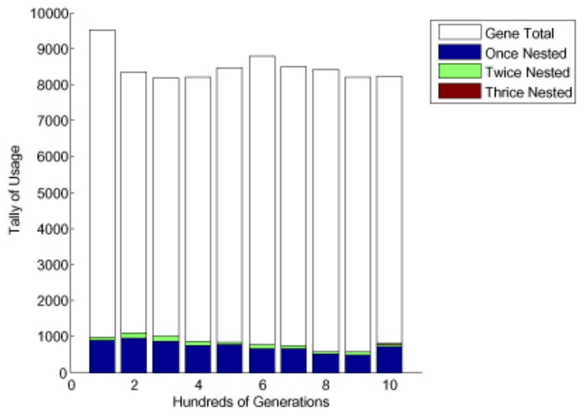


Figure 10: Structure Genes Used by the Top 10% of Each Generation During a Unsuccessful Evolution (Once Nested at Bottom)

Finally, statistics looking at the structure of the rules are examined. The actions of every reading frame within a used gene is tallied for each run. Its important to note that the sum of these tallies will be high than the total number of genes used because nested genes contain multiple reading frames. Furthermore, while the actual numbers are given, it is the relative ratios that remain consistent among similar runs.

When looking at the actions used, we see that making a connection was the most common. This is to be expected as most of what the ANN does is grow connections. However, we see the second most common action was the end turn action, which prevents the network from performing tasks. This shows that the control of growth is about as important as growth itself. In other words, evolving rules prohibiting actions maybe as important as involving rules that promote actions.

Table 4: Actions in Executed Genes

	Make Connection	Make Node	Do Nothing	End Turn
Successful Run	4297	1628	1566	3981
Unsuccessful Run	12750	4346	612	8054

6. CONCLUSION

This paper follows the evolutionary path of the genome. Furthermore, it shows differences between qualities that good and poor evolution will have. One may be able to use these differences to actually help promote better results from evolution. Namely, punctuated evolution events seem to be critical for evolution to be a success. This would suggest that it maybe beneficial to periodically cause mass-extinctions in order to promote overall betterment of the population.

Another surprising outcome is the importance of growth-limiting rules in all evolution. One of the advantages to restricting growth it restricts the search space, usually leading to more efficient evolutions. However, it is difficult to know how much limiting is necessary *a priori* and allowing this retardation of growth to evolve along with the population is obviously beneficial, even for poor evolutionary paths.

Further work will include a study of the type of conditionals used for IF-THEN tests, comparisons where evolution is unsuccessful for different reasons, a look at the structure of the unused portions of the genome, and check the scalability of the genomes for large neural networks.

7. ACKNOWLEDGMENTS

The simulations were performed on GARUDA, a high-performance computing cluster hosted by the Civil Engineering Department at Caltech.

8. REFERENCES

[1] P. Bentley and S. Kumar. Three ways to grow designs: A comparison of embryogenesis for an evolutionary design problem. In *Genetic and Evolutionary Computation Conference*, pages 35–43, 1999.

[2] D. Federici and K. Downing. Evolution and development of a multicellular organism: Scalability, resilience, and neutral complexification. *Artif. Life*, 12(3):381–409, 2006.

[3] D. Floreano, P. DÃijrr, and C. Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 2008.

[4] F. Gruau. Genetic synthesis of modular neural networks. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 318–325, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[5] H. Kitano. Designing neural networks using genetic algorithms. *Complex Systems*, 4(4):461–476, 1990.

[6] P. Koehn. Genetic encoding strategies for neural networks, 1996.

[7] S. Ohno. *Evolution by gene duplication.* Springer-Verlag, 1970.

[8] J. B. Pollack, G. S. Hornby, H. Lipson, and P. Funes. Computer creativity in the automatic design of robots.

[9] A. Roy, E. Antonsson, and A. Shapiro. Genetic programming of an artificial neural network for robust control of a 2-d path following robot. In *ASME Design Engineering Technical Conferences*, New York City, NY, USA, 2008.

[10] K. O. Stanley, D. B. Ambrosio, and J. Gauci. A hypercube-based indirect encoding for evolving large-scale neural networks. *Artificial Life*, 2009.

[11] G. Theraulaz and E. Bonabeau. Coordination in distributed building. *Science*, 269(5224):686–688, August 1995.

[12] I. G. Tsoulos, D. Gavrili, and E. Glavas. Neural network construction using grammatical evolution. *IEEE International Symposium on Signal Processing and Information Technology*, pages 827–831, 2005.