Using Feedback in a Regulatory Network Computational Device

Generative and Developmental Systems

Rui L. Lopes
Centro de Informática e Sistemas da Universidade de Coimbra
Polo II - Pinhal de Marrocos
3030-290 Coimbra, Portugal
rmlopes@dei.uc.pt

Ernesto Costa
Centro de Informática e Sistemas da Universidade de Coimbra
Polo II - Pinhal de Marrocos
3030-290 Coimbra, Portugal
ernesto@dei.uc.pt

ABSTRACT

The relationship between the genotype and the phenotype in Evolutionary Algorithms (EA) is a recurrent issue among researchers. Based on our current understanding of the multitude of the regulatory mechanisms that are fundamental in both processes of inheritance and of development in natural systems, some researchers start exploring computationally this new insight, including those mechanism in the EA. The Artificial Gene Regulatory (ARN) model, proposed by Wolfgang Banzhaf was one of the first tentatives. Following his seminal work some variants were proposed with increased capabilities. In this paper, we present another modification of this model, consisting in the use the regulatory network as a computational device where feedback edges are used. Using two classical benchmarks, the n-bit parity and the Fibonacci sequence problems, we show experimentally the effectiveness of the proposal.

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic Methods

General Terms

Algorithms

Keywords

regulation, network, development, parity, Fibonacci

1. INTRODUCTION

Nature-inspired algorithms are used today regularly to solve learning, design and optimization problems, giving rise to a new research area called Evolutionary Computation (EC) ([6]). There are many variants of a basic algorithm that we can describe in simple terms: (1) randomly define an initial population of solution candidates; (2) select, according to fitness, some individuals for reproduction with variation; (3) define the survivors for the next generation; (4) repeat steps (2) and (3) until some condition is fulfilled. Typically, the objects manipulated by the algorithms are represented at two different levels. At a low level, the genotype, the representations are manipulated by the variation operators; at a high level, the phenotype, the objects are evaluated to determine their fitness and are selected accordingly. Because of that, we need a mapping between these two levels. The issue of the relationship between the genotype and the phenotype is as old as the area of itself, many claiming that the standard approach is too simplistic.

Today we are aware of the crucial role of regulation in the evolution and development of complex and well adapted creatures ([3]). Regulation is a consequence of external (i.e., epigenetic) or internal (i.e., genetic) influences. Some computational explorations have been proposed to deal with the inclusion of regulatory mechanisms into the standard evolutionary algorithm ([5, 2, 17]). Wolfgang Banzhaf ([1]) proposed an artificial gene regulatory (ARN) model and showed how it could be used computationally in different settings ([4, 16]). More recently [14] a variant of the ARN was proposed that solved some weaknesses of that model, by transforming the regulatory gene network into a computable tree-like expression as we do in standard GP. In this paper, we complement and enhance this latter proposal by making possible the use of feedback edges (equivalent to delays in an iterative process). The effectiveness of this proposal was experimentally tested and statistical validated with two classical benchmarks, namely the N-bit parity and the Fibonacci sequence problems. We clearly establish the generalization capabilities of the solutions produced.

The paper is organized as follows. Section 2 describes the ARN model as proposed by W. Banzhaf. Then, Section 3 describes our contributions, elucidating how we extract a program from a network and the extension to the previous model. In Section 4 we briefly refer to the problems used followed by the experimental setup in Section 5. The results are presented and analyzed in Section 6. Finally, in Section 7 we draw some conclusions and present some ideas for future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'11, July 12–16, 2011, Dublin, Ireland.
Copyright 2011 ACM 978-1-4503-0557-0/11/07 ...$10.00.

2. BANZHAF'S ARN

Genome

The Artificial Regulatory Network (ARN) ([1]) is composed of a binary genome and proteins. The genome can be generated randomly or by a process of duplication with mutation, also called DM, that is considered the driving force for creating new genes in biological genomes and as an important role in the growth of gene regulatory networks [18]. In the latter case we start with a random 32-bit binary sequence, that is followed by several duplication episodes. As we will see later the number of duplications is an important parameter. The mutation rate is typically of 1%. So, if we have 10 duplication events then the final length of the genome is $2^5 \times 2^{10} = 32768$. The genome is divided in several regions, namely a regulatory site, the promoter and the gene itself. The first 32 bits of the regulation zone are the enhancer site, while the following 32 bits are the inhibitory site. The promoter is located downstream and has the form $XYZ01010101$. This means that only the last 8 bits are fixed. A gene is a five 32-bit long sequence, i.e., a 160-bit string.

Gene expression

The genotype - phenotype mapping is defined by expressing each 160-bit long gene, resulting in a 32-bit protein. This correspondence is based on using a majority rule: if we consider the gene divided into 5 parts of size 32 each, at position i, say, the protein's bit will have a value corresponding to the most frequent value in each of these 5 parts, at the same position. Figure 1 gives an idea of the representation.

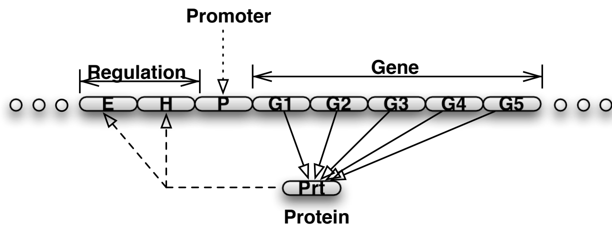

Figure 1: Artificial Regulatory Network, after W. Banzhaf

Regulation

The proteins can bind to the regulatory region. The strength of the binding is computed by calculating the degree of complementarity between the protein and each of the regulatory regions (enhancer, e_i, or inhibitory,h_i), according to formula 1:

$$x_i = \frac{1}{N}\sum_{j=1}^{N} c_j e^{\beta(\mu_j - \mu_{max})} \quad (1)$$

where x_i can the the enhancer or the inhibitory region, N is the number of proteins, c_j the concentration of protein j, μ_j is the number of bits that are different in each position of the protein and of the regulation site, μ_{max} is the maximum match achievable, and β is a scaling factor. The production of a protein along time depends on its concentration, which in turn is a function of the way it binds to the regulatory regions. It is defined by the differential equation

$$\frac{dc_i}{dt} = \delta(e_i - h_i)c_i$$

.

Genes interact mediated by proteins. If, say, gene **A** expresses protein p_A and that protein contributes to the activation of gene **B**, we say that gene **A** regulates **B** (see figures 2 and 3).

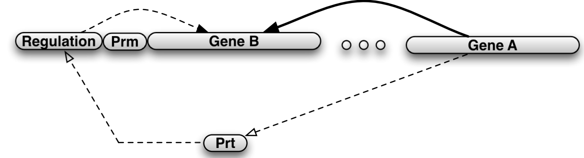

Figure 2: Gene - Protein - Gene interaction

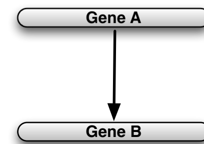

Figure 3: Gene - Protein - Gene interaction

Notice that in order for a link to exist between any two genes, the concentration of the corresponding protein must attain a certain level, and that depends on the strength of the binding.

Computational Device

Using this process we can build for each genome the corresponding gene regulatory network. These networks can be studied in different ways. We can be concerned by topological aspects (i.e., to study the degrees distribution, the clustering coefficient, small world or scale free, and the like) or the dynamics of the ARNs (i.e., attractors, influence of the protein-gene binding threshold) ([15], [4]). This is interesting, but from a problem-solving perspective what we want is to see how the model can be used as a computational device. In order to transform an ARN into a computational problem-solver we need to clarify what we put in the system (including the establishment of what is the relationship with the environment) and what we extract from the system. At the same time we need to define the semantics, that is, the meaning of the computation in which the network is engaged. Finally, and as a consequence of the points just identified, it is also fundamental to determine if we are interested in the input/output relationship or if what we want is the output. A solution for the latter situation was proposed in [12] in the context of optimization problems. The idea is to define (randomly) two new contiguous 32-bit sequences in the genome. The first one being a new inhibitory site, and the second one a new activation site. All generated proteins can bind to these sites. The levels of activation and of inhibition can be computed as before (see equation 1) , but there is no gene (thus no protein) attached (see figure 4).

The state of this site is just the sum of all bindings (see equation 2) and is defined as the output. This additional

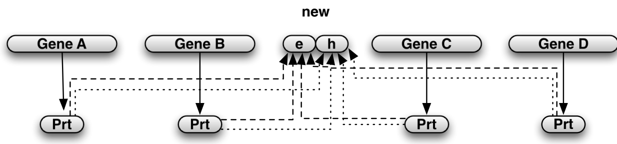

Figure 4: Fabricating an output

binding is thus a method to extract a meaning from the variation of the proteins' concentrations along time.

$$s(t) = \sum_i (e_i - h_i) \quad (2)$$

To use the model as a representation formalism for genetic programming one needs to define what are the inputs and what are the outputs. For that purpose the ARN model was extended in two directions ([16]). First, some extra proteins, not produced by genes but contributing to regulation, were introduced and act as inputs. Second, the genes were divided into two sets, one producing proteins that are used in regulation (i.e., transcriptional factors), and a second one with proteins without regulatory function which are used as outputs. These two types of genes are distinguished by having different promoters (see figure 5).

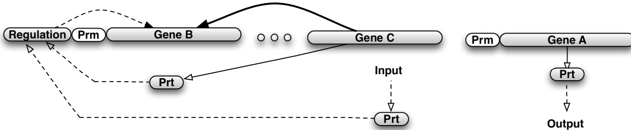

Figure 5: The modified ARN

This model was tested with the classic control problem known as the pole balancing problem.

3. REGULATORY NETWORK COMPUTATIONAL DEVICE (RENCODE)

With the model just described we have to define, more or less arbitrarily, what are the inputs and what are the outputs. To overcome this limitation [14] used the ARN architecture as genomic representation for a new computational model, described in the remaining of this section. The main idea is to simplify the ARN to produce a computable tree-like expression similar to a GP tree (although with the introduction of feedback the result is a cyclic graph). Besides that, in order to greatly increase the efficacy and the performance of the system, special variation operators, besides mutation, were also introduced.

This model was tested successfully with three benchmark problems, different from the ones presented in this article. In order to solve the problems presented in Section 4 some modifications are necessary, namely the inclusion of feedback connections, as will be clarified at the end of this section.

Extracting Circuits from ARNs.

The networks resultant from ARN genomes are very complex, composed of multiple links (inhibition and excitation) between different nodes (genes). In order to extract a circuit from these networks they must first be reduced, input and output nodes must de identified, and we must ascribe a semantic to its nodes. The final product will be an executable feed-forward circuit.

Listing 1: The reduction algorithm.

```
1  def reduce(network):
2    for each gene in network:
3      replace bindings by edge (e-h)
4        if (e - h ) <= 0:
5          remove edge
6    for each edge(i,j) in new network:
7      if  edge(i,j) < edge(j,i):
8        remove edge(i,j)
9      else:
10       remove edge(j,i)
```

Listing 1 shows the pseudocode for the reduction algorithm. We start by transforming every pair of connections, excitation (e) and inhibition (h), into one single connection with strength equal to the difference of the originals (e-h) (step 3). Every connection with negative or null strength will be discarded (steps 4-5). Then a directed graph is built adding only the nodes with active edges and the strongest edge between each pair of nodes (steps 6-10). This process is illustrated in Figures 6 to 8.

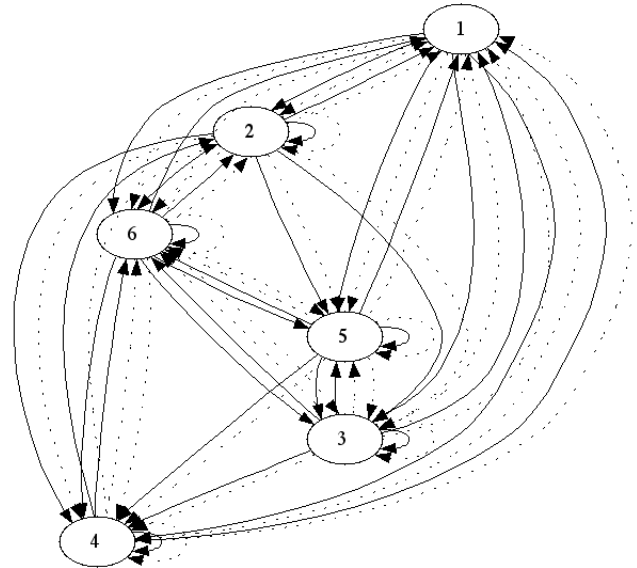

Figure 6: ARN example. The dotted and full edges represent, respectively, inhibition and excitation relationships. The numbers are just identifiers.

Next, the output node is chosen: the one with the highest connectivity. After that the circuit is built backwards from the output function until the terminals (nodes without input edges) are reached. If, at any point of this process, there is a deadlock (every function is input to some other), again the gene with highest connectivity is chosen, discarding its role as input to others, and thus resulting in a feed-forward circuit of *influences* between genes.

To complete the process, a mapping is needed linking nodes (i.e., genes) to functions and terminals. To that end

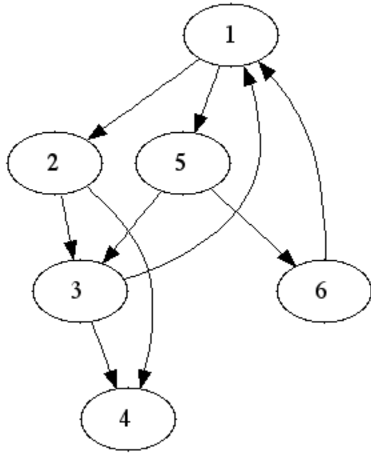

Figure 7: The network from Fig. 6 after application of the algorithm in List. 1.

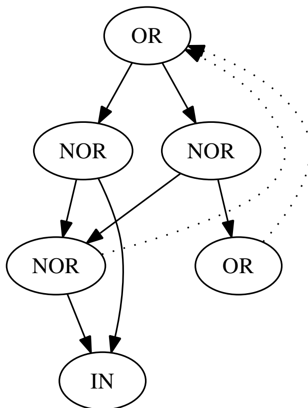

Figure 8: The resulting circuit from the reduced network depicted in Fig. 7. The nodes represent gates or the input stream (IN). The output of the circuit is taken from the top OR. The binary string is streamed through the IN node. Functions take their inputs from the nodes where their out-edges point to. The dotted edges represent feedback connections.

we use the gene-protein correspondence using the protein's signature to obtain the function/terminal by a majority vote process. As an example, to code the function set `{ +, -, *, / }` only two bits are needed. These are obtained by splitting the 32-bit protein into sixteen 2-bit chunks and applying the majority rule in each position of the chunks. This will provide us with an index to the function set. If some determined problem has more than one input, for instance four, then the majority rule is applied again over the terminal protein signature (its binary stream) to define to which input it corresponds.

Finally, in order to improve the evolvability of the genomes, the authors proposed also variation operators inspired by the concepts of transposons and non-coding DNA, which can delete/copy a part of the genome (*transposon-like*), or introduce non-coding (*junk*) genetic material (streams with 0s) in the genome. The results reported show an efficiency increase when using any of the operators with small lengths (versus fixed size genomes with only the mutation operator), although not statistically significant in every case. Moreover, in one of the problems, the *junk* operator did not improve the results. The average results also show that the *transposon-like* operator is more *stable*, since the standard deviation values tend to be much lower in this case. Based on these observations, the experiments described in this article will use only the *transposon-like* operator.

Feedback.

To solve problems like symbolic regression and the santa-fe ant trail (see [14]), where there is no need for feedback, each time a node is chosen and the corresponding function is added to the circuit, the inputs from functions already added are simply discarded. This results in state-less feed-forward circuits, which are not adequate for problems where some kind of memory is needed.

On the other side, when a node takes input from a node (function) already in the circuit, it is possible to use it as feedback (instead of discarding it), resulting in a state-full feed-forward circuit (please refer to Fig. 10 for an example), similarly to a cyclic graph. This one-level of indirection allows to save information from the previous state(s) when the circuit is iterated, as will be demonstrated by its application to two different problems, described in the next section.

4. PROBLEMS

Two problems were tackled taking advantage of feedback connections in the regulatory network computational device: the n-bit parity and the Fibonacci sequence. The next subsections describe both problems and the experimental setup is presented at the end.

4.1 N-bit Parity

The N-bit parity problem is a typical benchmark in Evolutionary Computation (EC). The goal of this problem is to evolve a boolean function (or circuit) that takes a binary string as input and returns a single output which indicates wether the number of 1s in the string is even (0) or odd (1). The typical function set is `{AND, OR, NAND, NOR}`, using the input bits $\{x_1, x_2, ..., x_n\}$ as terminal set.

It has been recognized as a difficult problem for evolutionary systems by different authors [10, 7]. The traditional GP method is to directly evolve circuits (represented by trees). This approach does not scale well though, 5-bit parity so-

lutions are usually very difficult to evolve, and many times unsuccessful [10]. There have been improvements to this representation, amongst others the use of automatically defined functions [11], but still were not tackling n-bit parity.

Modern developmental systems have been proposed that solve the n-bit parity problem. In Self-Modifying Cartesian Genetic Programming [9], the solutions are programs that construct circuits. In [13] an artificial development system is evolved that also is capable of growth to generate a circuit that outputs the parity bit of any binary string. In the latter case however, the function set used was composed of multiplexers, making the task easier.

When the aim is to find a general solution it is a common approach to use recursion or to iterate over the bits of the binary input string, producing solutions suitable for any input size [19, 20]. As described in Sect. 3 it is possible to use feedback connections in ReNCoDe, creating state-full circuits. Making use of this feature, the current work will evolve circuits which iterate over the input string bits, similarly to recursive approaches, generating the parity bit as output.

The 3-bit parity problem is used as the fitness function. That is, the evolutionary process halts when a solution that generates the correct parity bit for the eight (2^3) input combinations is found (if unsuccessful it terminates when the maximum number of evaluations is reached). After the evolutionary process finds a solution for the 3-bit parity this is tested for generalization, to a maximum of 24-bit input streams (see Section 6.1).

4.2 Fibonacci Sequence

The Fibonacci sequence is a recursive sequence that obeys the rules defined in Eq. 3 to 5:

$$F(0) = 0 \tag{3}$$

$$F(1) = 1 \tag{4}$$

$$F(n) = F(n-1) + F(n-2) \tag{5}$$

This sequence has many real-world applications, for instance, in financial markets's analysis and computer algorithms. Moreover in can be found in nature in diverse forms, such as branching in trees or the arrangement of a pine cone.

This problem was first solved in genetic programming using recursive tree structures [10]. The results found in literature for this problem show that obtaining general solutions is not an easy task. Most of the approaches take a very high number of evaluations to find a solution and are not completely effective. Moreover the generalization is usually poor, although few approaches managed to obtain good generalization results (for a good summary of these results please refer to [8]).

The approach followed here is to evolve circuits using the arithmetic operators as the function set `{+,-,*,/}` and `{0,1}` as the terminal set. Protected division is used, returning 0 whenever the denominator is 0. It was noticed that the resulting programs do not have terminal nodes (see Section 6), but this is an evolutionary choice not a design one. The circuits are then iterated to produce the sequence elements. The fitness function is the amount of correct numbers for the first ten elements of the Fibonacci sequence, in contrast to other approaches where the first 12 and the first 50 elements are used for training [8]. The evolutionary process ends when the first ten elements are correctly generated or when the maximum amount of evaluations is reached.

Finally, the circuits are tested for generalization over the first 74 elements of the Fibonacci sequence, in order to compare with the results reported in [8].

5. EXPERIMENTAL SETUP

A standard evolutionary strategy was used for the experiments: $(10+100)-ES$. As in previous implementations of ReNCoDe, there is no crossover but a transposon-like operator is used, which can copy or delete a portion of the genome, therefore making the genome's length variable. The mutation operator is always applied. The number of runs and their parameterizations are summarized in Table 1. The parameters were chosen by trial and error during testing. No sensitivity tests nor parameter optimization were realized.

Parameter	Value
Number of Runs	100
Initial Population	100
Max. Evaluations	10^5
Number of DMs	5
DM Mutation Rate	0.01
Mutation Operator Rate	0.01
Protein Bind Threshold	16
Genome Length	Variable
Operator Type	Transposon
Operator Length	80

Table 1: Evolutionary Parameters Values

6. RESULTS AND ANALYSIS

In this section the results obtained will be presented and discussed. An argumentation on the proof of generality is also presented for the n-bit parity problem.

6.1 N-bit Parity

The results for the parity problem are summarized in Table 2. The algorithm efficiently solves the 3-bit parity problem in every run. The results are promising, as few as 900 evaluations are required to find a solution. However, the distribution is quite large as the standard deviation is close to the average and as we can see from the box-plot of the runs (Fig. 9). After terminating evolution the circuits were tested with bit-streams with maximum size 24 bits, achieving a very high success rate.

From the values on the number of functions and connections we can see that there is not much bloat in the solutions. As an example the smallest circuit found in the 100 runs is shown in Fig. 10. Each node represents either a gate or the binary string input. The functions get their input values from the nodes pointed out by the out-edges. The dotted edges represent the feedback connections (because it is a feed-forward circuit, the value of the function was not updated yet and keeps the last result). Feedback connections which have not been initialized hold 1.

Generality.

Testing a circuit for input strings bigger than 24 bits is computationally very expensive. The generality of the circuits

% of Successful Runs	100
% of General Solutions	98
Min. Numb. of Evaluations	900
Avg. Numb. of Evaluations (Std. Dev.)	16094 (12652)
Min. Numb. of Functions/Connections	5 / 7
Avg. Numb. of Functions/Connections	6 / 9

Table 2: Summary of the n-bit parity results for 100 runs.

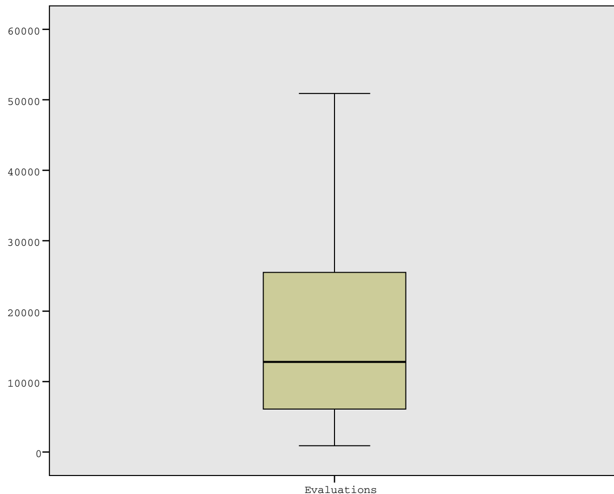

Figure 9: Distribution of the number of evaluations for each run of the n-bit parity problem.

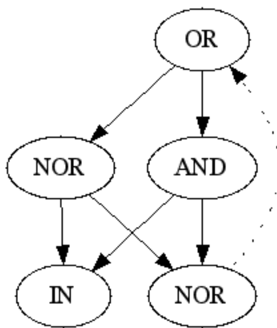

Figure 10: Smallest circuit found that generates the parity of any string. The nodes represent gates or the input stream (IN). The output of the circuit is taken from the top OR. The binary string is streamed through the IN node. Functions take their inputs from the nodes where their out-edges point to. The dotted edges represent feedback connections.

was tested with maximum 24-bit streams, however it can be proven analytically that it is correct for n-bit size. The logic expression of the circuit can also be used to analyze the solution. Taking as an example the circuit presented in Fig. 10, it will be proven that the result is suitable for any input size.

Let x_i be the current input bit, x_{i-1} the output of the previous iteration (initialized with 1), and o_i the output that will be generated. We can then write the logic expression that translates the circuit (Eq. 6) and the corresponding truth table (Table 3).

$$b_i = \neg(x_i \vee \neg x_{i-1}) \vee (x_i \wedge \neg x_{i-1}) \tag{6}$$

where $x_0 = 1$ and $i = 1, 2, ..., n$.

x_{i-1}	x_i	o_i
1	1	0
1	0	1
0	1	1
0	0	0

Table 3: Truth table of the circuit.

The truth table of the circuit shows that it is equivalent to the XOR of the current input bit with the previous output. After iterating the complete input string, the result is the EXOR of all the input bits, with the initialization making the XOR with 1, which is the definition of parity (see Eq. 7).

$$F = 1 \oplus \bigoplus_{i=0}^{n-1} x_i \tag{7}$$

6.2 Fibonacci Sequence

The Fibonacci sequence problem is an interesting one. From the results that can be found in the literature, it is hard to evolve systems that generalize well.

In the case of the model presented here every run was successful in finding a solution for the first ten elements, in the given time. The average effort (number of evaluations) results are show in Table 4. The amount of successful runs corresponds to the percentage of runs that found a solution for the first ten elements. The percentage of general solutions indicates how many of these generalize correctly for 74 elements. In Table 5 we transcribe the results from Self-Modifying Cartesian Genetic Programming (SM-CGP) [8], as a reference for the performance. Only the results using the first twelve elements as fitness function are presented, since in the current article we use only the first ten. As a note, the function set used here is a small subset of the one described in [8]. Also in this case the distribution of the effort taken in each run is quite large (Fig. 11).

Figure 12 shows the smallest circuit found that generates the Fibonacci sequence. Each function takes its input values from the nodes pointed to by its out-edges. Dotted edges represent feedback connections. The circuit is then iterated updating the functions in a bottom-up fashion. In each iteration the result at the top node is retrieved as an element in a sequence. It can be observed that the evolutionary process does not need any terminal node. This may be consequence of the functions being initialized with the value 1, but nevertheless the sequence starts with 0.

% of Successful Runs	100
% of General Solutions	96
Min. Numb. of Evaluations	8600
Avg. Numb. of Evaluations (Std. Dev.)	62504 (46157)
Min. Numb. of Functions/Connections	6/9
Avg. Numb. of Functions/Connections	7/13

Table 4: Summary of the Fibonacci sequence results for 100 runs.

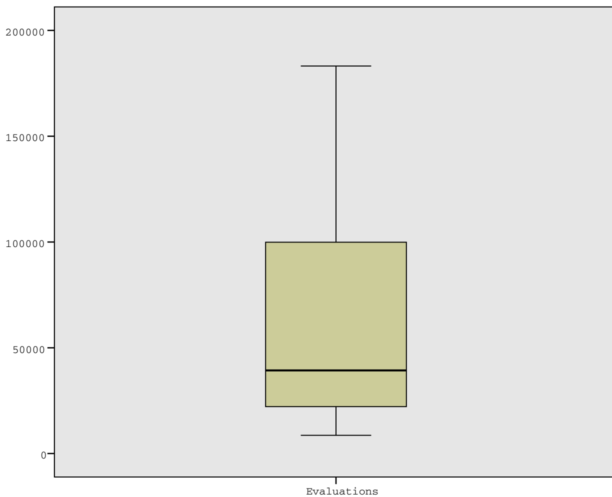

Figure 11: The distribution of the number of evaluations values for each run of the Fibonacci problem.

% of Successful Runs	89.1
% of General Solutions	88.6
Avg. Numb. of Evaluations	1019981

Table 5: Summary of the Fibonacci sequence results for SM-CGP[8], based on 287 runs. The first 12 elements of the sequence are used as fitness, and `{0,1}` as the starting condition.

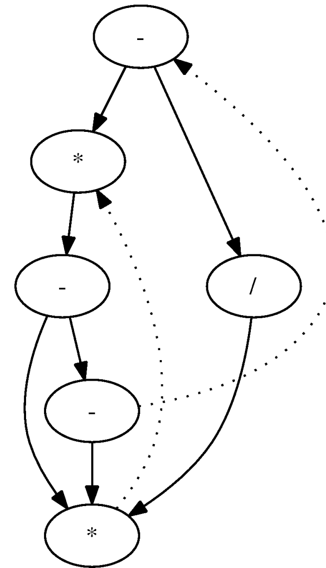

Figure 12: Smallest circuit found that generates the Fibonacci sequence. The nodes represent arithmetic operators. The output of the circuit is taken from the top. Each iteration the nodes are updated from bottom to top and the result is taken as an element of the sequence. Functions take their inputs from the nodes where their out-edges point to. The dotted edges represent feedback connections.

7. CONCLUSIONS AND FUTURE WORK

In this article the Regulatory Network Computational Device was extended to solve a different class of problems, where state-full circuits are necessary. A method of using the feedback connections in the artificial regulatory networks was introduced and the efficacy of the approach was demonstrated by solving two benchmark problems, the n-bit parity and the the Fibonacci sequence (see Section 4).

The algorithm proved to be capable of evolving n-bit parity generators as well as Fibonacci sequence generators, always finding a solution for the problems in a short amount of time, and showing good generalization capabilities. For the n-bit parity a circuit was taken as example and the generality of this was proven analytically. For the Fibonacci sequence the achievements were compared with others in the literature, showing some improvements.

Future work on this computational device should include sensitivity tests of the evolutionary parameters, namely, the number of initial DM events, the DM mutation rate and the operator length which appear to be problem dependent and influence both the performance (the amount of required evaluations) and the success rates of the algorithm.

8. REFERENCES

[1] W. Banzhaf. Artificial regulatory networks and genetic programming. *Genetic Programming Theory and Practice*, pages 43–62, 2003.

[2] J. Bongard. Evolving modular genetic regulatory networks. In *IEEE 2002 Congress on Evolutionary Computation (CEC2002)*, pages 1872–1877. IEEE Press, 2002.

[3] E. H. Davidson. *The regulatory genome: gene regulatory networks in development and evolution.* Academic Press, 2006.

[4] P. Dwight Kuo, W. Banzhaf, and A. Leier. Network topology and the evolution of dynamics in an artificial genetic regulatory network model created by whole genome duplication and divergence. *Bio Systems*, 85(3):177–200, 2006.

[5] P. Eggenberger. Evolving morphologies of simulated 3D organisms based on differential gene expression. In P. Husbands and I. Harvey, editors, *Fourth European Conference of Artificial Life*. MIT Press, 1997.

[6] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer Verlag, 2003.

[7] C. Ferreira. Genetic representation and genetic neutrality in gene expression programming. *Advances in Complex Systems*, 5(4):389–408, 2002.

[8] S. Harding, J. Miller, and W. Banzhaf. Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing. *Genetic Programming*, pages 133–144, 2009.

[9] S. Harding, J. F. Miller, and W. Banzhaf. Developments in Cartesian Genetic Programming: self-modifying CGP. *Genetic Programming and Evolvable Machines*, 11(3-4):397–439, June 2010.

[10] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[11] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs (Complex Adaptive Systems)*. The MIT Press, 1994.

[12] P. Kuo, A. Leier, and W. Banzhaf. Evolving dynamics in an artificial regulatory network model. *Lecture Notes in Computer Science*, pages 571–580, 2004.

[13] T. Kuyucu, M. A. Trefzer, J. F. Miller, and A. M. Tyrrell. A scalable solution to n-bit parity via artificial development. *Research in Microelectronics and Electronics*, pages 144–147, 2009.

[14] R. L. Lopes and E. Costa. ReNCoDe : A Regulatory Network Computational Device. In S. Silva and J. Foster, editors, *EuroGP2011, Lecture Notes in Computer Science, vol 6621*, volume 6621, pages 142—-153, 2011.

[15] M. Nicolau and M. Schoenauer. Evolving specific network statistical properties using a gene regulatory network model. In G. and others Raidl, editor, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 723–730, Montreal, 2009. ACM.

[16] M. Nicolau, M. Schoenauer, and W. Banzhaf. Evolving genes to balance a pole. *European Conference on Genetic Programming*, 6021:196–207, 2010.

[17] D. Roggen, D. Federici, and D. Floreano. Evolutionary morphogenesis for multi-cellular systems. *Genetic Programming and Evolvable Machines*, 8(1):61–96, Dec. 2006.

[18] S. a. Teichmann and M. M. Babu. Gene regulatory network growth by duplication. *Nature genetics*, 36(5):492–6, May 2004.

[19] M. L. Wong. Evolving recursive programs by using adaptive grammar based genetic programming. *Genetic Programming and Evolvable Machines*, 7(1):127, Mar. 2006.

[20] M. L. Wong and K. S. Leung. *Evolving recursive functions for the even-parity problem using genetic programming*, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.