STATISTICALLY NEUTRAL PROMOTER BASED GA FOR EVOLUTION WITH DYNAMIC FITNESS FUNCTIONS

F. Bellas and R. J. Duro
Grupo de Sistemas Autónomos
Universidade da Coruña
Spain
fran@cdf.udc.es, richard@udc.es

Abstract

In this paper we consider the use of promoter genes and introns in the encoding of variable length artificial neural network structures for their evolution. To support these genotypes we present the adaptation of a structured genetic algorithm we have called Promoter Based Genetic Algorithm (PBGA) to contemplate the evolution of the architecture and weight values of artificial neural networks which regulates the expression of the different genes in the chromosome in a statistically neutral manner. Obviously, this leads to a non direct genotype-phenotype transformation which becomes very efficient in dynamic environments. We study some examples where the advantages of using this type of representation over traditional genetic algorithms in problems with changing fitness functions become evident.

Keywords

Genetic algorithms, artificial neural networks, variable length genotype, phenotype construction, promoter genes.

1. Introduction

A very serious problem in standard Genetic/Evolutionary algorithms is that they tend to converge towards homogeneous populations, that is, populations where all of the individuals are basically the same. In static problems this would not be a drawback if this convergence took place after reaching the optimum. Unfortunately, there is no way to guarantee this, and diversity may be severely reduced long before the global optimum is achieved.

Dynamic problems, that is, problems where the fitness function varies in time, are even more challenging. If the population has converged to a given solution, and there is no genetic diversity, the only way to follow a changing fitness function is through mutation, which is basically a very slow random search.

Three approaches may be considered to try to solve this problem. Two of them are based on exogenous actions over the population and the third one acts over the transcription from genotype to phenotype. In the first group, some authors have resorted to methods where individuals were introduced in the population when necessary in order to generate diversity. These individuals were generated randomly, as in the case of [1][2], or through some heuristic implying that at certain points of the algorithm a secondary memory would preserve interesting individuals that would be injected in the population as needed [3]. In both cases it is necessary to somehow monitor diversity.

Other researchers have considered restrictive selection mechanisms within the algorithms so as to enforce a given level of genetic diversity in the populations using concepts like similarity between individuals in a given metric and which, as in the previous approach, require the definition of a consistent distance measure in the population [4].

Finally, a third more biologically inspired way of addressing the problem has been to tweak with the representation of the individuals and their genotype-phenotype transformation through the selective expression of genes.

It is interesting to note that despite the fact that nature makes extensive use of gene expression techniques, they have not often been used in the realm of evolutionary algorithms. The situation is changing and we are seeing an increase in the study of these techniques in the last few years [5][6][7][8].

There exist two basic biologically based approaches to gene expression, Diploid representations and promoter based mechanisms. Diploid genotypes are made up of a double chromosome structure where each strand contains information for the same functions. Whenever a phenotype is constructed from the genotype, one of the two possible alleles for each gen is chosen following a dominance mechanism which may change with time. As not all of the genes making up the chromosomes are expressed, and as the fitness of an individual is determined by its phenotype, the recessive genes are shielded from selective pressure thus providing a memory within the encoding of the genotype. These techniques were introduced in computational evolution by Goldberg [9] who claimed that a diploid representation combined with a dominance map can outperform a standard evolutionary algorithm in dynamic problems. Several studies and applications of these mechanisms such as those by Ng and Wong [6] and Ryan [5] may be found in the literature. In this work, however, we will concentrate on the other gene shielding/expression mechanism, that is, the use gene

promoters and, consequently, loosely speaking, of introns as unexpressed pieces of genetic code. Before we go into how we implement this approach, let us provide an overview of the biological facts that underpin it.

2. Some biological facts

In prokaryotes (bacteria and other simple cells) all the DNA coding for a protein is continuous. In more complex, eukaryotic, cells, however, the encoding DNA is generally discontinuous: sequences of encoding DNA (exons) are interspersed with long sequences of non-encoding DNA. This non-encoding DNA sequences, usually about 10-fold longer than the exons, are called introns and even though for a long time they have been considered "junk", the fact that they are so common and have been preserved during evolution leads many researchers to believe that they serve some function.

To control where a protein is encoded, the chromosome contains protein begin and protein end signals called codons. A codon is a group of three bases - A, T, C, or G - and codes for a single amino acid. A start codon is made up of the letters ATG, which codes for the amino acid methionine. When the machinery of the cells sees that first ATG, it knows that the instructions for making a protein begin at this point. The code is always read in groups of three, so the start codon also gives the cell's machinery it's so-called reading frame. Each set of three letters thereafter corresponds to a single amino acid. A stop codon tells the cell's machinery that it has reached the end of the protein and should stop translating the code. Stop codons come in three different forms - TGA, TAG, and TAA.

It must also be considered that almost every cell in an organism has a copy of every single gene the whole organism needs. Different genes are expressed in cells corresponding to different organs. Obviously, one would not want a gene coding for toes to be expressed in the lungs. Gene promoters are in charge of controlling these effects. Gene promoters are important regulatory structures that control the initiation and level of transcription of a gene. They sit upstream of the gene and dictate whether, or to what extent, that gene is turned on or off.

3. Enhancing a GA through a modified genotype

In the very brief introduction presented above we have provided an indication of the elements that should go into genotype encoding in order to allow for gen expression. These elements are exons and introns, gene promoters and codons. Using these elements the "cell machinery" that is, the part of the GA in charge of constructing the phenotype will know how to make the final organism from the genotype. If these elements are an intrinsic part of the encoding, they will provide for an unobtrusive way of evolving what is expressed and how, and thus make the operation of the algorithm much smoother.

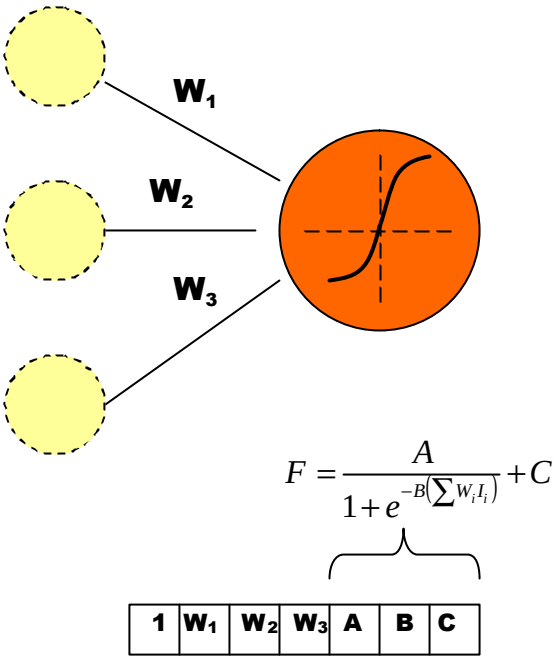

Figure 1: Sigmoid type neuron with inbound connections which constitutes the basic unit. The first field corresponds to the gene promoter value, in this case it is 1 and the neurone will be included in the phenotype.

To achieve this objective we have considered a GA that evolves variable size feedforward artificial neural networks (ANNs). These neural networks, in our case, are encoded into sequences of genes for constructing a basic ANN unit. Each of these blocks is preceded by a gene promoter acting as an on/off switch that determines if that particular unit will be expressed or not. In order to simplify the algorithm, we have decided to make use of these gene promoters also as start and end codons due to the position they occupy in the chromosome.

As basic unit we have taken a neuron with all of its inbound connections (figure 1). Consequently, the genotype of a basic unit is a set of real valued weights followed by the parameters of the neuron (in this case a traditional sigmoid) and preceded by an integer valued field that determines the promoter gene value and, consequently, the expression of the unit. By concatenating units of this type we can construct the whole network. For the sake of clarity, in the examples presented here, a maximum of ten neurons per layer and four layers were considered (input, two hidden and an output layer). This obviously does not mean that the networks will be 10-10-10-10 networks, as their size will depend on how many and which genetic units are expressed. Consequently we can have anything from a 0-0-0-0 network to a 10-10-10-10 network. The important aspect about this encoding is that whatever is not expressed is still carried by the genotype in evolution but it is shielded from direct selective pressure. Therefore, we are establishing a clear

difference between the search space and the solution space. In addition, a given weight genotype can yield many different phenotypes depending on the status of the promoter genes.

The general idea is based on some of the encoding concepts used for the Structured Genetic Algorithm (sGA), developed by Dasgupta and McGregor [10] as a general hierarchical genetic algorithm. They applied a two level interdependent genetic algorithm to solving the knapsack problem and developing application specific neural networks [11]. A two layer SGA was used to represent the connectivity and weights of a feed-forward neural network. Higher level genes (connectivity) acted as a switch for sections of the lower level, weight representation. Sections of the weight level whose corresponding connectivity bits were set to one, were expressed in the phenotype. Those whose corresponding bits had the value of zero were retained, but were not expressed.

4. Crossover and mutation

The only problem we may have in the algorithm is how to perform crossover and mutation without being extremely disruptive or generating a bias in the evolution of what genes are expressed. We must bear in mind that we are crossing over not only weight values, but the architecture of the network. Consequently, one has to be careful about how disruptive crossover or mutation will be on the information units found in the genotype.

If we take two parent chromosomes, they will probably not have the same expressed neurons, and these are the ones that directly affect the fitness of the individual when implemented in the phenotype. Thus, we find two types of information in the parent genotypes that must be recombined in order to produce an offspring: genes corresponding to expressed neuron units, which are responsible for fitness, and genes for unexpressed neurons, about which we have little information. Thus, as there is no real order within a layer, it makes sense to rearrange the chromosomes of the parents so as to first insert all of the expressed neurons and then the unexpressed ones for each layer. After rearranging the chromosomes, we can perform a special type of unit by unit crossover, that is, the first basic unit of parent A is crossed over with the first basic unit of parent B, the second with the second, and so on. The resulting offspring unit will have a 25% chance of being a copy of parent A´s unit, a 25% chance of being a copy parent B´s unit and a 50% chance of being the result of a one point crossover between the two parental units. The reason for doing this is that we provide a chance of preserving whole working units and, in a certain sense, permit simultaneously carrying out a coarse grained crossover at the basic unit level (50% chance) and a fine grained crossover at the individual gene level (50% chance). To be statistically neutral with respect to the expression of the genes, now we have to carefully perform a crossover operation with to the promoter genes that control the expression of the gene sequences. This crossover follows a rule whereby if both parent units are expressed, the offspring unit is expressed, if both parent units are not expressed, the offspring unit is not expressed and finally, when one unit is expressed and the other is not, there is a 50% chance of the offspring unit being expressed. Thus, on average, we maintain the number of expressed units and prevent a bias in this term.

Regarding mutation, things are simpler, and the only consideration we must make is that gene promoters must be mutated with different frequency from that of regular genes. Note that mutating gene promoters may be very disruptive as it affects in a very serious way the composition of the phenotype, whereas mutation of the rest of the genes is on average much more gradual on

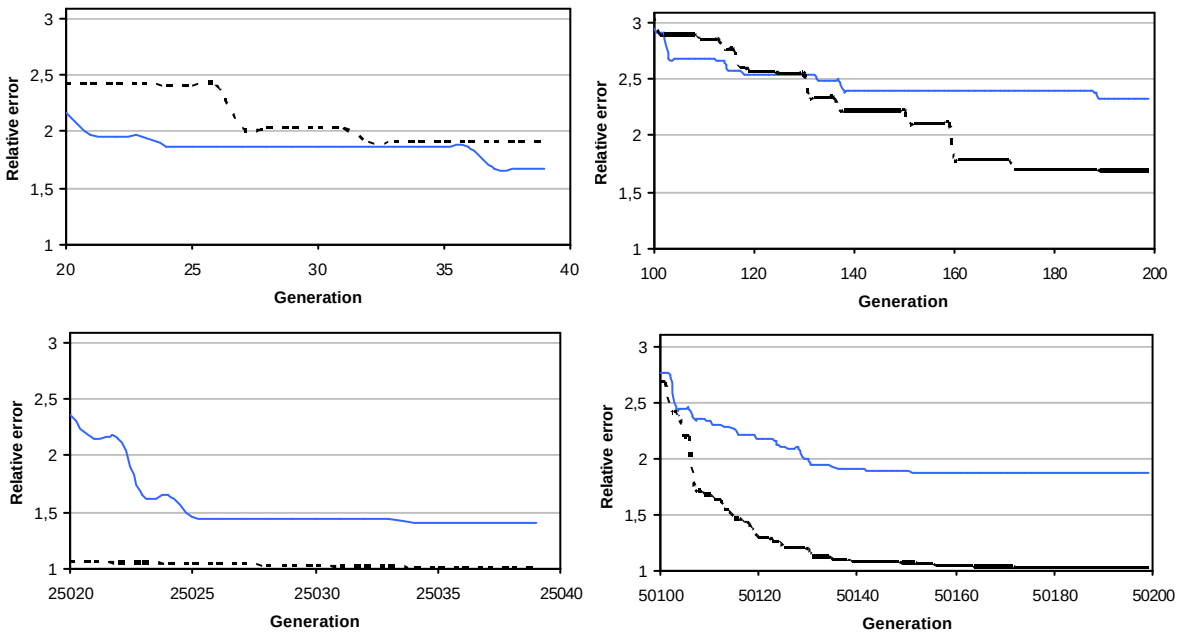

Figure 2: the left hand side we have graphs corresponding to switching between two fitness functions every 20 generations and on right hand side the switch is performed every 100 generations. The top graphs correspond to the evolution in the first switch, the bottom ones to the evolution after many switches

the resulting phenotype, especially if these mutations take place in unexpressed areas of the genotype. Consequently, we decided to use a much lower mutation rate on the gene promoters and a non linear (cubed random) mutation mechanism for the real valued genes.

5. Some experimental results

To test the performance of this type of strategy, we have considered the evolution of artificial neural networks that perform very simple functions. We made use of a standard Genetic Algorithm for comparison and a Promoter Based Genetic Algorithm with the extensions mentioned above. Both GAs make use of a simple tournament selection scheme and the mutation and crossover parameters were the same. The genotype encoding in the standard genetic algorithm was the same as in the promoter based one except for the fact that there were no gene promoters and no unexpressed information was preserved in the genotype. The networks in the standard Genetic Algorithm grew or decreased in size by addition of units with random parameters and the deletion of basic units.

In order to test the amount of information that is stored in the promoter genes, we use a given fitness function for N generations of evolution and different one for the next N generations and keep cycling between the two until we stop. We expect the Promoter Based Genetic Algorithm to converge faster as the iterations (fitness function switch cycles) progress, because some of the previously learned information has a chance of remaining in the unexpressed part of the genotype.

Figure 2 displays the results obtained for the two algorithms in the first cycle and after a few thousand fitness function switches. In this case we switched between two fitness functions corresponding to neural networks that modelled the sum of the inputs and networks that modelled the product of the inputs. The figure contemplates two cases. In the left hand side we have graphs corresponding to switching between two fitness functions every 20 generations and on right hand side the switch is performed every 100 generations. The top graphs correspond to the evolution in the first switch, the bottom ones to the evolution after many switches (the x axis displays the number of generations that have been run). The dashed line in each graph indicates the PBGA and the solid line the standard one. All the data have been normalized to the best solution (whose error we indicate by 1). Several things can be seen in this figure. First, in the first cycle, both GAs perform similarly, especially at the beginning. After many cycles, it can be observed that the PBGA obtains much better results. This is especially noticeable in the case where we only perform 20 generations of evolution between fitness function switches. As the chromosomes do not have time to converge, the genotype preserves the necessary information and the PBGA obtains the best result after two generations of evolution, whereas the standard GA needs basically the same number of generations as in the first cycle. In the case where we run 100 generations before switching, the result is very similar, but now, as the phenotype population has more

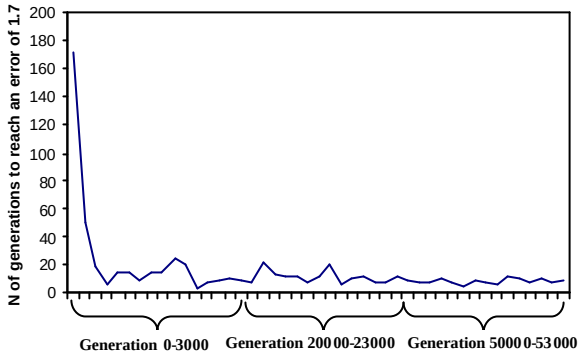

Figure 3: Number of generations required by the PBGA in order to achieve a given error value (1.7 normalized error) as cycles progress. The x axis shows the tendency in different periods of the evolution.

time to converge, it takes the PBGA a little longer to achieve the best results. Despite this longer time, it is still a lot faster than the standard GA and the final error obtained is almost halved.

In figure 3 we display the number of generations required by the PBGA in order to achieve a given error value (1.7 normalized error) for the same fitness function when this function appeared interspersed with other fitness functions. In this case, two fitness functions were alternated. Each one was used for 200 generations before switching. It is clear from the graph that the first time the PBGA sees the function it needs 170 generations to achieve the error level desired. The second time it takes it only 50 generations and by the fourth it only takes 6 generations to obtain the desired error level. This result clearly indicates that the preservation of information in the unexpressed part of the genotype really leads to dramatic improvements in the speed of evolution towards a solution the system has seen totally or partially before.

The way the system is switching between learnt representations can be appreciated in figure 4, where we display de average number of neurons that make up the networks in the population throughout evolution for switching every 100 generations between a fitness function for modelling f(x)=x*sin(x)+y*sin(y) with an ANN and a fitness function for modelling f(x)=x*y.

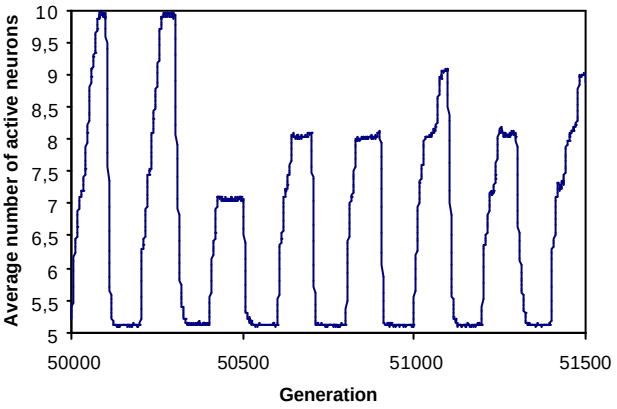

Figure 4: Average number of neurons per ANN in the population for the evolution using a PBGA when switching every 100 generations between a fitness function for modelling f(x)=x*sin(x)+y*sin(y) and a fitness function for modelling f(x)=x*y.

There are clearly two average sizes. One of them corresponds to the minimum possible size chosen by the network to solve the problem indicated by fitness function 2 which is around 5 neurons and the other size to that chosen for the harder function, which is between 8 and 9 neurons. It is interesting to note how the PBGA basically switches neurons on and of very fast when a transition between fitness functions occurs. Note that this number is the number of neurons active in the network, but it does not provide any indication of the distribution of neurons between layers. In fact, some solutions are obtained where the network only uses one hidden layer and others where two hidden layers are obtained.

6. Discussion

From the figures we have presented it is clear that allowing the genotypic representation of the organism to carry more information that is necessarily expressed in the phenotype results in a great advantage in terms of preserving genotypic diversity in the population even when the phenotypes have converged to a solution. When this occurs, the expressed parts of the genotypes tend to become very similar, thus increasing the frequency of the genes that help to achieve the solution within the population. When the fitness function changes, the genetic algorithm will be able to keep on working appropriately by drawing from the diversity present in the unexpressed parts of the genotype in order to modify the phenotypes and follow the new fitness function.

What is very relevant now is that a sort of genetic memory is created within the genotype due to the high probability of those useful genes that were successful in previous generations, and consequently appeared more frequently in the population, to become a part of the unexpressed parts of the genotype. The immediate result of this memory is that when a fitness function that has been seen before (or which requires combinations of basic units that were used in previous successful runs) is contemplated again, the GA achieves the desired phenotype much faster than before. In the simple tests presented in the previous section the solution was reached more than 20 times faster.

Another advantage of this type of encoding is that phenotypes can grow or decrease in size depending on the processing required for each fitness function. The appropriate number of neurons will be selected in order to achieve the desired goal. This provides a very simple mechanism for obtaining variable size neural networks on one hand and as a consequence, it allows the GA to select the necessary inputs to perform the function it needs to carry out without having to compensate for redundant or unnecessary information. This is a very important point as it is much more difficult for a neural net to compensate for an input than to simply turn off that input neuron, which is basically what this type of evolution allows.

Summarizing, this mechanism provides for a clear separation between the search and solution space, that is, genotypic and phenotypic spaces, and consequently lack of diversity in the latter does not imply this lack former. It also permits a preservation of functionality while allowing a continuation of search, something that is very difficult when using traditional GAs evolving variable length ANNs. In the traditional case, any neuron pruning or neuron addition is a very dramatic change which usually leads to undesired results.

7. Conclusions

In this work we have presented an implementation of a statistically neutral promoter based genetic algorithm. This algorithm is characterized by the fact that there is a transcription between genotype and phenotype regulated by a set of promoter genes. These promoters determine if a given gene is expressed or not. Those genes that are not expressed do not participate in the phenotype and are thus shielded from overselection, allowing for a mechanism for preserving diversity in populations as well as implementing a memory of blocks that were useful in for previous fitness functions. In the examples included in this paper we have seen that these algorithms perform much better than equivalent traditional GAs in developing ANNs for problems where the fitness function changes over time. In addition, the encoding used provides for a very simple mechanism to operate with variable length ANNs.

8. Acknowledgements

This work was supported by the MCYT of Spain through project TIC2000-0739C0404.

References

[1] J. Grefenstette, Genetic algorithms for changing environments, In Mnner, R. and Manderick, B. (editors). *Parallel Problem Solving from Nature 2*. Amsterdam: North Holland. 1992, 137-144.

[2] R.J. Duro, J. Santos and A. Sarmiento, GENIAL, an Evolutionary Recurrent Neural Network Designer and Trainer. *Lecture Notes on Computer Science*. V 1105, 1996, 295-301.

[3] P. Hartono and S. Hashimoto, Migrational GA that preserves Solutions in Non-Static Optimization Problems, *Proc. of IEEE-SMC 2001*, 2001, 255-260.

[4] J. Kubalik, L. J. M. Rothkrantz, Genetic Algorithm with Limited Convergence. *Proceedings 6th Joint Conference on Information Sciences*. 2002, 61-614.

[5] Conor Ryan, J. J. Collins, Polygenic Inheritance - A Haploid Scheme that Can Outperform Diploidy. *PPSN 1998*, 178-187.

[6] Khim Peow Ng and Kok Cheong Wong, A new diploid sceme and dominance change mechanis m for nonstationary function optimisation. *In Proceedings of the Sixth International Conference on Genetic Algorithms*, 1995, 159-166.

[7] H. Kargupta and K. Sarkar, Function Induction, Gene Expression, and Evolutionary Representation Construction. *Proceedings of the Genetic and Evolutionary Computation Conference*. vol 1, 1999, 313-320.

[8] Vesselin K. Vassilev, Julian F. Miller, The Advantages of Landscape Neutrality in Digital Circuit Evolution. *ICES 2000*, 2000, 252-263.

[9] D. E. Goldberg and R. E. Smith. Nonstationary function optimization using genetic algorithms with dominance and diploidy. In J. J. Grefenstette, editor, *Second International Conference on Genetic Algorithms*, 1987, 59-68.

[10] Dipankar Dasgupta and Douglas R. McGregor. A structured genetic algorithm: The model and first results. *Computer Science Department IKBS-2-91, University of Strathclyde, Glasgow, U.K.*, 1991.

[11] Dipankar Dasgupta and Douglas R. McGregor. Designing application-specific neural networks using the structured genetic algorithm. In *COGANN-92 Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks*, Baltimore, MD, 1992. 87-96.