Polymorphy and Hybridization in Genetically Programmed Networks

Arlindo Silva[1], Ana Neves[1], Ernesto Costa[2]
Centro de Informática e Sistemas da Universidade de Coimbra

[1] Escola Superior de Tecnologia, Instituto Politécnico de Castelo Branco,
Av. do Empresário, 6000 Castelo Branco – Portugal
{arlindo, dorian}@dei.uc.pt
[2] Departamento de Engenharia Informática, Universidade de Coimbra, Polo II – Pinhal de Marrocos, 3030 Coimbra – Portugal
ernesto@dei.uc.pt

Abstract. In this paper we discuss the polymorphic abilities of a new distributed representation for genetic programming, called Genetically Programmed Networks. These are inspired in a common structure in natural complex adaptive systems, where system functionality frequently emerges from the combined functionality of simple computational entities, densely interconnected for information exchange. A Genetically Programmed Network can be evolved into a distributed program, a rule based system or a neural network with simple adjustments to the evolutionary algorithm. The space of possible network topologies can also be easily controlled. This allows the fast exploration of various search spaces thus increasing the possibility of finding a (or a better) solution. Experimental results are presented to support our claims.

1. Introduction

As artificial agents grow in intelligence and autonomy it becomes harder to design by hand the controllers to guide those agents. Evolutionary approaches to the synthesis of autonomous intelligent agents have been proposed mainly in the evolutionary robotics and artificial life communities. These approaches try to reduce the intervention of the agent designer by evolving a solution from a population of individuals using the principles of natural selection instead of more complex engineering techniques.

Evolutionary approaches to agent synthesis can be divided in three main groups according with the representation used for the individuals:

- Neural networks [4, 8, 10].
- Rule based systems [6, 7, 9].
- Computer programs [1, 2].

The choice of the most appropriate controller architecture for autonomous agents is the center of an ongoing discussion [2, 15] which will probably never end. But from it we can conclude that the use of very specific representations has obvious disadvantages in the fact that when the approach fails to find a solution to a given problem, a

lot of effort is needed to start over, using a new representation and corresponding approach. Even if a solution is found, it would sometimes be useful to have an easy way of trying other representation, to see if better solutions could be found in the corresponding search space.

In this paper we address the way the polymorphy in Genetically Programmed Networks (GPN), i.e. the ability to be evolved into individuals with dissimilar structures and topologies, can be used as a tool for the fast exploration of different search spaces.

Genetically Programmed Networks were initially presented as a new evolutionary approach to the synthesis of controllers for autonomous agents and showed promising results in benchmark problems [16,17,18]. They are the result of our search for an evolutionary algorithm centered on a polymorphic representation. This search has been, from the beginning, conditioned by three self-imposed constraints:

- First, the approach should be as independent from the problem as possible to reduce the designer intervention, thus incorporating the other evolutionary approaches main advantage.
- Second, it should provide support for the emergence of memory mechanisms, which we believe are essential for the agent's autonomy and intelligence. This aspect of our approach has been discussed in [17].
- Third, the level of polymorphy in the representation should be enough to allow the evolution of controllers with the most common architectures in previous approaches, namely neural networks, rule based systems and computer programs.

These constraints have been fulfilled in GPN while maintaining the flexibility and simplicity characteristic of evolutionary approaches.

Our representation was inspired in natural complex adaptive systems, which, as noted in [11], are frequently composed of simple computational entities densely interconnected, with the connections allowing the flow of information between entities. This structure is recurrent in nature, being seen in our brain and immune system as well as in cities or ecosystems. The same model is also used in several computational systems, e.g. artificial neural networks and cellular automata.

It seemed to us that the flexibility of the underlying structure could provide us with a representation capable of sustaining the desired polymorphy. The way memory mechanisms are implemented in systems with this structure, by allowing delays in the connections used to transmit information, e.g. recurrent connections in neural networks, also suggested an elegant way to of fulfilling our second constraint.

Genetically Programmed Networks are presented in more detail in the second section of this paper. The evolutionary algorithm and the control of polymorphy are discussed in section 3. Section 4 is devoted to the presentation of experimental results to illustrate the polymorphic abilities of GPN. Finally, in section 5, we summarize our work and draw some conclusions.

2. Genetically Programmed Networks

We can follow the natural model that served us as inspiration, to describe the representation used for Genetically Programmed Networks. We will start by a description of the GPN genotype, followed by a discussion of its phenotype and the relation between them.

2.1. GPN Genotype

In GPN **entities** are represented by programs that completely codify their behavior, which is simply a set of computations. Entities are usually called nodes in a GPN.

Connections are represented by variables in the program that codifies an entity's behavior. When a program accesses one of these variables we say that a connection is being established between the entity associated with the program and the entity corresponding to the variable. This means we don't need an explicit representation for connections, thus avoiding the usual problems with variable length representations when there is the need to insert or remove a given connection.

We now need a representation for the **programs**. Instead of using the most usual representation in evolutionary algorithms, more specifically in classical genetic programming (GP) [13], which is an explicit syntactic tree-based representation, we chose to use a linear representation as suggested in [12]. This allowed us some savings in computational resources, i.e. memory and processor time, at the cost of a somewhat more complex implementation. Each program is simply a vector of integer indexes each one referencing a function, variable or constant in corresponding tables. As in GP, functions, variables and constants can be chosen from *a priori* defined sets. The differences between these sets and the ones used in GP will be discussed later.

We still have a syntactic tree but is no more an explicit one, being instead implicitly constructed when the indexes in the chromosome are recursively interpreted as functions, variables and constants.

The **genome** of a GPN individual is then nothing more than a sequence of programs codified as integer vectors. An individual will have **n chromosomes**, corresponding to **n programs**, each associated with one of **n entities** or nodes**.**

2.2. GPN Phenotype

In terms of phenotype, a GPN individual is a network of interconnected entities or nodes. Besides nodes and connections a GPN has two other components:

1. **Inputs**, which allow the GPN to access information about the environment. The agent, after gathering information with its sensors, is responsible for presenting the GPN inputs with the relevant information, preprocessed or not.
2. **Outputs.** The final computations of a GPN are copied to a set of outputs so that the agent can use them to choose an action to perform. The entities whose final computations are copied to the outputs are called **external nodes**, while the other ones are called **internal nodes**.

As already mentioned, the connections in a GPN are created when a variable corresponding to the output of an entity is accessed by a program associated with other entity. Following the same principle, connections between inputs and entities are established when a variable corresponding to an input is used in the entity's program. Two types of connection can be created:

1. **Forward connections** are established from an entity *q* to an entity *p* when the program associated with *p* uses a variable f_q which contains the output of the entity *q* in the same iteration. In the same way, a forward connection between an input *m* and an entity *p* is established when a variable i_m with the value of input *m* is used by the program associated with *p*. Forward connections between entities correspond to **functional** relations, since the destination entity will use the result computed by the origin entity as a partial result in its own computations.

2. **Recurrent connections** are established from an entity *q* to an entity *p* when the program associated with *p* uses a variable r_q which contains the value computed by *q* in the **previous** iteration. Recurrent connections result in the delayed transmission of information, thus permitting the emergence of memory mechanisms in a GPN, i.e. they allow the formation of **temporal** relations between entities. This type of connection is not permitted between an input and an entity.

A special type of forward connections is assumed to always exist from external nodes to the GPN outputs. Its function is only to allow the instantaneous flow of the values computed by the programs associated with external nodes to the correspondent GPN outputs.

3. The Evolutionary Algorithm

Since a GPN is merely a sequence of programs it seems obvious that a form of genetic programming should constitute the ideal evolutionary algorithm to evolve a population of GPN individuals. In fact, the algorithm centered on Genetic Programmed Networks is a new flavor of distributed GP, with several differences from classical GP to deal with the particularities of our representation.

The most important differences deal with the choice of function and terminal sets for the programs and the inclusion of a new level of operators.

3.1. Controlling Polymorphy

In opposition to GP the most important aspect in the choice of functions and terminals is not if they are the most suitable for the problem being solved, but if they allow the evolution of the type of controller we want.

To better control the polymorphy of our representation the function set is divided in two other sets:

1. The **root set** *R*, which has the functions that are allowed to be chosen as the root of the program's syntactic tree.
2. The **function set** *F*, which has all the functions that can be included in the program except at its root.

Internal and external nodes have different root, function and terminal sets, respectively *Ri*, *Fi*, *Ti* and *Re*, *Fe*, *Te*.

Manipulating the terminal sets we can control the topology of the individuals, i.e. the space of allowed connection patterns that can be explored. Table 1 presents the terminal sets for three different topologies.

Complete Topology
T_i={ i_1, i_2, ..., i_l, r_1, r_2, ..., r_m, r_{m+1}, r_{m+2}, ..., r_{m+n}, ...}
T_e={ i_1, i_2, ..., i_l, f_1, f_2, ..., f_m, r_1, r_2, ..., r_m, r_{m+1}, r_{m+2}, ..., r_{m+n}, ...}
One Layer Topology
T_i={ i_1, i_2, ..., i_l, r_1, r_2, ..., r_m, r_{m+1}, r_{m+2}, ..., r_{m+n}, ...}
T_e={ i_1, i_2, ..., i_l, r_1, r_2, ..., r_m, r_{m+1}, r_{m+2}, ..., r_{m+n}, ...}
Memory Topology
T_i={ r_1, r_2, ..., r_m, r_{m+1}, r_{m+2}, ..., r_{m+n}, ...}
T_e={ i_1, i_2, ..., i_l, f_1, f_2, ..., f_m, ...}

Table 1. Terminal sets for three different topologies, assuming a network with *l* inputs, *m* internal nodes and *n* external nodes.

In a complete topology every possible connection is allowed. This is the most flexible topology, but corresponds to the larger search space, since is the one that includes more terminals. In a one layer topology, forward connections are not allowed from internal nodes to external nodes. This topology is less flexible than a complete one, but produces the GPN with fastest execution, since all programs can be ran concurrently. Finally, a memory topology allows recurrent connections only to internal nodes, which act as a memory layer. This is the topology with the smallest search space, but is also the least flexible.

Manipulating the function and root sets, we can control the architecture of the GPN being evolved. When no particular restrictions are made, all function and root sets are equal and we say that distributed programs are being evolved. To evolve rule based systems an *if-than-else* function is the only member of the root sets, and its not allowed in the function sets. This way all evolved programs will be *if-than-else* rules. To evolve neural networks more significant changes are needed. Every terminal gains an associated randomly generated weight, the root sets can only have a transfer or activation function and the function sets only members are an *add* and a *subtract* function. Each evolved program is no more than the sum of weighted terminals passed trough the activation function.

These examples maintained equal root and function sets for internal and external nodes. The networks evolved when this is true are called homogeneous networks. If the sets are chosen differently for internal and external nodes, we can evolve hetero-

geneous GPN, e.g. neural networks with different activation functions in the internal and external nodes, and even hybrid controllers, e.g. neurons in the internal nodes and rules in the external nodes.

Distributed Program
$F_i=F_e=R_i=R_e=\{...\}$
Rule Based System
$R_i=R_e=\{if\text{-}then\text{-}else\}$
$F_i=F_e=\{...\}$
Neural Network
$R_i=R_e=\{transf\}$
$F_i=F_e=\{add, subtract\}$

Table 2. Function and root sets for three different architectures, assuming a network with *l* inputs, *m* internal nodes and *n* external nodes.

We used these examples since they correspond to the most common architectures in other evolutionary approaches, but it is obvious that with further manipulations other architectures (and topologies) could be used.

The functions and terminals that are not architecture or topology related are chosen to allow the computations that take place in the nodes and do not directly depend on the problem. Functions or terminals with side effects, common in GP, are not allowed in our approach, since they are highly problem dependent.

3.2. Operators

At program level we maintained the use of the three most important operators in GP, namely fitness proportional reproduction, tree crossover and tree mutation. A new level of GPN operators had to be introduced to control the program operators' application strategies. In the experiments whose results we present here, three of these operators are used:

1. GPN reproduction: when a GPN is selected for fitness proportional reproduction, all the programs of the new individual are copied from the corresponding node in its parent.
2. GPN mutation: when a GPN is selected for mutation, the resulting offspring will be obtained by applying program mutation to every program in the parent individual.
3. GPN crossover: when two GPN are selected for crossover, the two resulting offspring will be obtained by applying program crossover to every pair of corresponding programs in the parents.

Other strategies are obviously possible, e.g. instead of always applying program crossover to each pair of programs, it could be applied with some chosen probability.

3.3. Evolving Genetically Programmed Networks

The evolutionary algorithm is an obvious instance of the generic evolutionary algorithm and similar to GP. After the creation of an initial population by randomly generating $m \times n$ programs, where m is the population size and n is the number of nodes in an individual, the main evolutionary cycle begins:

- All individuals are evaluated by being allowed to control the agent in the given task.
- A temporary population is constituted by selecting individuals with replacement from the population using a tournament selection strategy.
- GPN crossover is applied to 85% of the temporary population, GPN reproduction to 10% and GPN mutation to the remaining 5%, obtaining a new population of offspring.
- The population is completely substituted by the new population and a new cycle begins.

The cycle ends when some stopping condition is satisfied, usually when a solution is found or a limit generation is reached.

4. Experimental Results

To test the polymorphic abilities of Genetically Programmed Networks, we chose a benchmark problem with a fitness landscape considered hard for traditional GP. This is the well known "Ant in the Santa Fé Trail" problem, which had its first variant "Ant in the John Muir Trail" proposed by Collins [5]. In the version most used in GP an artificial ant has to follow a discontinuous trail of food pellets in a toroidal cellular world. The ant has only one sensor that informs it of the presence or not of food in the next cell and can choose from three actions: move forward, turn left while remaining in the same cell, turn right while remaining in the same cell. The ant can use a maximum of 400 actions to complete the trail.

Since Koza [13] applied GP to this problem for the first time, that it has been considered a GP hard problem, not because it is difficult to find a solution, but because GP compares poorly to other search procedures in the effort it needs to find one. This effort is the measure we use here for comparison and equals the number of individuals that must be evaluated so that a solution is found with a probability of 99% [13].

Langdon [14] discusses the difficulties GP finds in this problem in terms of its fitness landscape and presents some results on the effort needed by several GP an non GP approaches to find a solution. We summarize these results in Table 3 where it can be seen that all GP based approaches need significantly more effort that simple hill climbing. The best results obtained by an evolutionary approach, also shown in Table 3, are the ones presented in [3] and use a form of evolutionary programming. But even these are only marginally better than the ones obtained with hill climbing.

Approach	Effort
Genetic Programming [14]	450,000
Sub-Tree Mutation [14]	426,000
Parallel Distributed GP [14]	336,000
Hill Climbing [14]	186,000
Evolutionary Programming [3]	136,000
Distributed Program / Complete Topology	88,800
Distributed Program / One Layer Topology	84,000
Distributed Program / Memory Topology	81,700
Rule System / Complete Topology	125,000
Rule System / One Layer Topology	134,400
Rule System / Memory Topology	85,000
Neural Network / Complete Topology	60,000
Neural Network / One Layer Topology	53,300
Neural Network / Memory Topology	39,000
Heterogeneous Neural Network / Complete Topology	42,300
Heterogeneous Neural Network / One Layer Topology	44,000
Heterogeneous Neural Network / Memory Topology	51,700
Rule System – Neural Network Hybrid / Complete Topology	81,000
Rule System – Neural Network Hybrid / One Layer Topology	96,000
Rule System – Neural Network Hybrid / Memory Topology	43,500
Distrib. Program. – Neural Network Hybrid / Complete Topology	150,000
Distrib. Program. – Neural Network Hybrid / One Layer Topology	96,000
Distrib. Program. – Neural Network Hybrid / Memory Topology	86,400

Table 3. Effort needed by several GP, non GP and GPN based approaches to solve the ant problem.

We hoped that by applying GPN to this problem we could find architecture / topology pairs corresponding to search spaces more friendly to genetic search. To verify this, we made 18 groups of 200 experiences corresponding to 18 architecture / topology pairs, including some hybrid forms. The experiments and results in terms of effort are summarized in Table 3. All the evolved GPN had two inputs, with values 10 when food was ahead and 01 otherwise. We used 6 internal nodes and 3 external nodes connected to 3 outputs. The output with the largest value decided which action to perform. The program size was limited to 64 so that the maximum of 9×64 instructions would allow a fair comparison with the other approaches, which usually have a 500 instructions limit. The evolution of distributed programs used root and function sets with the functions *and*, *or*, *not*, *equal*, *greater* and *if-than-else*. To evolve rule systems the *if-than-else* function was removed from the function sets and made the only member of the root sets. Neural networks were evolved as described before.

From Table 3 we can see that our best approach, a neural network / memory combination, needed only to evaluate 39,000 individuals to find a solution with 99% probability, which is roughly one third of the best non GPN approach and less than one fourth of the effort needed by hill climbing. All other combinations found solu-

tions with less effort than the needed by hill climbing and only one pair performed worst that the best non GPN approach. This shows the aptitude of our representation and methodology to solve this particular problem. It also shows that our claims towards the polymorphy of GPN can hold and that that polymorph can really be a useful tool in finding more friendly fitness landscapes to search, when dealing with hard problems.

While most combinations obtained good results it can be seen that the neural network architecture and the memory topology are usually associated with the better ones. While this was unexpected for the architecture, since we didn't had any clue on which should perform better, we cannot say the same for the topology since it is the one that uses less terminals while still allowing the emergence of the memory mechanisms needed to solve the problem. But it must be noted that other problems could possible need the flexibility of the complete topology or the faster execution of one layer GPN.

5. Conclusions

The usefulness of a polymorphic, problem independent representation for the evolutionary synthesis of controllers for autonomous agents was discussed. We proposed such a representation, called Genetically Programmed Networks, inspired in the structure of natural complex adaptive systems. We claimed that GPN could be used to evolve neural networks, distributed programs and rule-based systems with several topologies, and even hybrids of the listed architectures.

To support our claims, we presented the results of evolving GPN with 18 different pairs architecture / topology to solve a GP hard problem. The results show that GPN could be used to find instances of the general GPN representation corresponding to search spaces where our algorithm needed much less effort to find a solution than previous approaches.

We hope these results can be extended to other problems, allowing for the quick exploration of many promising search spaces, so that new or better solutions can be found, or at least that known solutions can be found faster.

6. Acknowledgements

This work was partially funded by the Portuguese Ministry of Science and Technology under the Program PRAXIS XXI.

References

1. W. Banzhaf, P. Nordin, and M. Olmer, **"Generating Adaptive Behavior for a Real Robot using Function Regression within Genetic Programming"**, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pp. 35-43, Morgan Kaufmann, 13-16 July 1997.

2. R. Brooks, "**Artificial Life and Real Robots**", *Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, MIT Press, Cambridge, MA, 1992, pp. 3-10.
3. K. Chellapilla, "**Evolutionary programming with tree mutations: Evolving computer programs without crossover**", *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 431-438, Morgan Kaufmann, 13-16 July 1997.
4. D. Cliff, P. Husbands and I. Harvey, "**Analysis of Evolved Sensory-Motor Controllers**", *Cognitive Science Research Paper*, Serial Nº CSRP 264, 1992.
5. R. Collins and D. Jefferson, "**Antfarm: toward simulated evolution**", *Artificial Life II, Santa Fe Institute Studies in the Sciences of the Complexity*, volume X, Addison-Wesley, 1991.
6. J. Donnart and J. Meyer, "**A Hierarchical Classifier System Implementing a Motivationally Autonomous Animat**", *Proceedings of the Third Int. Conf. on Simulation of Adaptive Behaviour*, MIT Press/Bradford Books, pp. 144-153.
7. M. Dorigo and U. Schnepf, "**Genetics-Based Machine Learning and Behaviour Based Robotics: A New Synthesis**", *IEEE Transactions on Systems, Man, and Cybernetics*, 23, 1, 141-154, January 1993.
8. D. Floreano and F. Mondada, "**Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot**", *From Animals to Animats III, Proc. of the 3rd Int. Conf on Simulation of Adaptive Behaviour*, MIT Press/Bradford Books, 1994.
9. J. Grefenstette and A. Schultz, "**An Evolutionary Approach to Learning in Robots**", *Proc. of the Machine Learning Workshop on Robot Learning, 11th International Conference on Machine Learning*, July 10-13, 1994, New Brunswick, N.J., 65-72.
10. I. Harvey, P. Husbands and D. Cliff, "**Issues in Evolutionary Robotics**", *Proceedings of SAB92, the Second International Conference on Simulation of Adaptive Behaviour*, MIT Press Bradford Books, Cambridge, MA, 1993.
11. J. Holland, "**Hidden Order – How Adaptation Builds Complexity**", Addison-Wesley Publishing, 1995.
12. M. Keith, "**Genetic Programming in C++: Implementation Issues**", *Advances in Genetic Programming*, pp. 285-310, MIT Press, 1994.
13. J. Koza, "**Genetic programming: On the programming of computers by means of natural selection"**, Cambridge, MA, MIT Press, 1992.
14. W. Langdon and R. Poli, "**Why Ants are Hard**", Technical Report CSRP-98-04, The University of Birmingham, School of Computer Science, 1998.
15. S. Nolfi, D. Floreano *et al*, "**How to Evolve Autonomous Robots: Different Approaches in Evolutionary Robotics**", *Artificial Life IV*, pp. 190-197, MIT Press/Bradford Books, 1994.
16. A. Silva, A. Neves and E. Costa, "**Evolving Controllers for Autonomous Agents Using Genetically Programmed Networks",** *Genetic Programming, Proceedings of EuroGP'99*, LNCS, Vol. 1598, pp. 255-269, Springer-Verlag, 26-27 May 1999.
17. A. Silva, A. Neves and E. Costa, "**Genetically Programming Networks to Evolve Memory Mechanisms",** *Proceedings of the Genetic and Evolutionary Computation Conference*, Vol. 2, p. 1448, Morgan Kaufmann, 13-17 July 1999
18. A. Silva, A. Neves and E. Costa, "**Building Agents with Memory: An Approach using Genetically Programmed Networks",** *In the Proceedings of the 1999 Congress on Evolutionary Computation (CEC 99)*, Washington, USA, 6-9 *Genetic and Evolutionary, Computation Conference*, Orlando, Florida, USA, 13-17 July, 1999. Morgan Kaufmann.