Evolving Controllers for Autonomous Agents Using Genetically Programmed Networks

Arlindo Silva[1]
arlindo@dei.uc.pt
http://www.dei.uc.pt/~arlindo

Ana Neves[1]
dorian@dei.uc.pt
http://www.dei.uc.pt/~dorian

Ernesto Costa[2]
ernesto@dei.uc.pt
http://www.dei.uc.pt/~ernesto

Abstract – This article presents a new approach to the evolution of controllers for autonomous agents. We propose the evolution of a connectionist structure where each node has an associated program, evolved using genetic programming. We call this structure a Genetically Programmed Network and use it to successfully evolve control systems with very different architectures, by making small restrictions to the evolutionary process. Experimental results of applying this method to evolve neural networks, distributed programs and rule-based systems all capable of solving a common benchmark problem, the Ant Problem, are presented. Comparisons with other known genetic programming based approaches, show that our method requires less effort to find a solution.

1 – Introduction

One of the many questions still without answer in the emergent field of evolutionary robotics concerns the choice of the most appropriate architecture to be evolved as the control system of the autonomous robots [Nolfi94]. The same problem poses itself in other areas where there is a need to evolve controllers for autonomous agents, e.g. in artificial life. In the literature we can find several approaches to this problem, the most promising of which seem to be:

- **Neural Networks** - This is undoubtedly the architecture more often chosen (and more strongly defended) to be evolved as the control system for autonomous agents [Cliff92] [Floreano94] [Harvey93]. The topology of the networks, however, varies substantially between approaches, and so does what is really evolved: connections' weights; weights and connections; weights, connections, and number of neurones, etc…
- **Programs** – Several authors propose the use of extended versions of genetic programming (GP) [Koza92] to evolve programs capable of controlling the robot. [Brooks92] suggests the use of GP with a high level *behavioural language*. [Bhanzaf97] uses GP to evolve assembly code, which maps sensorial inputs into actuator actions.
- **Rule Based Systems** - [Dorigo93] and [Grefenstette94] use several forms of classifier systems, or rule based systems, where the rules are genetically evolved to obtain valid controllers.

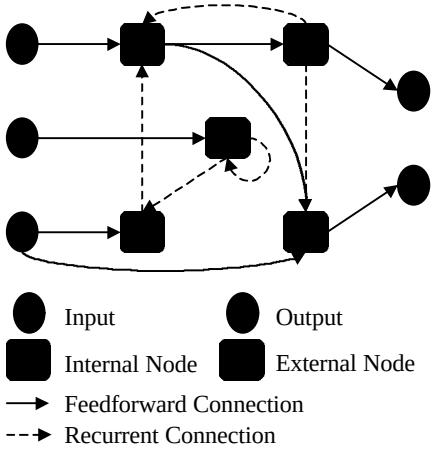


Figure 1: A Genetically Programmed Network. Every node has an associated program, generated by genetic programming.

In this paper we propose a new approach to controller evolution based on a connectionist structure we call Genetically Programmed Network (GPN). A GPN is constituted by a set of nodes, where each one node has an attached program, several connections between these nodes, a set of inputs and a set of outputs (see Figure 1).

The nodes are the computing elements in the network and each one uses the attached program to compute its output based on data flowing in from its connections. Connections act as a mean of transportation for data between the networks' inputs and nodes, from node to node and from nodes to the network's outputs. The network's inputs receive information from the agent, e.g. sensorial information, and make it available (by the existing connections) to every node in the network. Outputs present the result

[1] Escola Superior de Tecnologia, Instituto Politécnico de Castelo Branco, Av. do Empresário, 6000 Castelo Branco – Portugal
[2] Departamento de Engenharia Informática, Universidade de Coimbra, Polo II – Pinhal de Marrocos, 3030 Coimbra – Portugal

of the network computation to the agent to be used as a new command.

Programs are evolved using genetic programming in an evolutionary process that will, hopefully, produce a GPN capable of controlling the agent in a way it can achieve its goals. Genetic programmed networks are described in detail in section 2 and the evolutionary process used to evolve valid controllers from a GPN population is explained in section 3.

Section 4 describes the application of GPN to a benchmark problem, the Ant Problem. Controllers based on three different architectures, distributed programs, recurrent neural networks and rule-based systems, are successfully evolved using GPN and with better results than all other approaches known to us. In section 5 we draw some conclusions about this first results and we outline ongoing and future work in section 6.

2 – Genetically Programmed Networks

Genetically Programmed Networks were primarily designed to be used as controllers for intelligent agents. The role of the agent will be that of using its sensors to gather information about the environment and its actuators to act on it, while the GPN generates commands for the actuators based on the sensorial information available and its current internal state. The GPN itself can be seen as an agent, using its inputs as sensors and outputs as actuators. At a deeper level, the GPN has several nodes, each one with an associated program, inputs and one output. Each node can also be thought of as an agent, which makes a GPN an agent society were each member acts blindly for the common good: a high fitness value for the GPN individual. In this article we will use the word *agent* while referring to the entity being controlled by the GPN. When the term *environment* is used it is the environment where this entity dwells that is being mentioned.

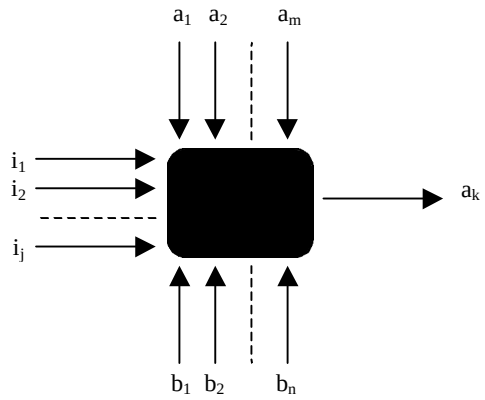
2.1 – GPN General Structure

A Genetically Programmed Network is constituted by (Figure 1):

- A set of inputs, whose values are received from the agent.
- A set of outputs, whose values are computed by the GPN and should be interpreted by the agent as a command.
- Two sets of nodes: internal nodes and external nodes.
- A set of connections, which link the previous components into a network.

Every node in the GPN is constituted by (Figure 2):

- A set of inputs.
- One output only.
- A genetically evolved program.



i_j - Network's input *j* at iteration *t*
a_m- Output of the node *m* at iteration *t*
b_n- Output of the node *n* at iteration *t-1*
P_k- Program associated with node *k*

Figure 2: A GPN node accepts inputs from the connections with other nodes and GPN inputs and computes its own output.

A node's input values at a certain iteration *t* can have three origins:

1. They can be the values of the inputs from the agent to the network at iteration t.
2. They can be the output values of other nodes in the network at iteration *t*.
3. They can be the output values of other nodes in the network at iteration *t-1*.

The structure of a particular GPN is partially defined at the start of the evolutionary process, since the networks' inputs and outputs, as well as the number of internal and external nodes are set *a priori*. Connections are developed as the programs associated with each node are evolved, so the structure will also be dependent on the evolutionary process.

2.2 - GPN Behaviour

The desired behaviour of a GPN is to compute, based on the inputs presented by the agent, an answer, codified in its outputs, which the agent can understand as a command. This command, or a sequence of commands, should lead the agent to perform as well as possible in some problem the environment poses to the agent. This behaviour is achieved by executing the programs associated with the network's nodes. These programs define the connections between nodes and compute each node output based on the inputs available to it.

2.2.1 – The Programs

The program associated which each node can be evolved using any variant of GP. The approach to GP we use is adapted from [Keith90] and it uses C++ instead of Lisp for extra speed and flexibility. What is particular to GPN is the terminal and function set. While in GP, typically, both of these sets are specifically chosen for the problem which is being solved, in GPN the restrictions are mainly architectural. What this means is that we choose terminals and functions accordingly with the architecture we want to evolve, i.e. evolving a neural network will require different functions and terminals than evolving a rule-based system. This point will be made clearer in section 4 when we apply our method to a specific problem.

The terminal set is of extreme importance, since it must contain a terminal for each input available to the node the program is associated with. This way a terminal set T can have the following terminals (see Figure 2):

$T=\{i_1, i_2, \ldots, i_l, a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_n, \ldots\}$

with i_j representing the value of the j^{th} network input, a_j representing the value of the j^{th} of a subset of m nodes whose outputs at iteration t this node has access to, and b_j representing the value of the output of the j^{th} node at instant *t-1*. This means that every node has access to every other node's output value at iteration *t-1*, but only to the output value of a subset of all nodes at instant *t*. This distinction is particularly relevant when it comes to differentiate between internal and external nodes. In fact, while external nodes can have a terminal set

$T_e=\{i_1, i_2, \ldots, i_l, a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_n, \ldots\}$,

internal nodes have a terminal set

$T_i=\{i_1, i_2, \ldots, i_l, b_1, b_2, \ldots, b_n, \ldots\}$,

which means they don't have access to any other node's output at iteration *t*.

Another difference between our system and GP is the fact that we use two function sets. Beside the usual function set *F*, whose members are all the functions that can be a node in the program, we use a special function set called the root set *R*. The members of this set are the functions that can be chosen to be the root node in the program's tree. When no particular restrictions are applied to the root node, the root set has the same members as the function set. But in the cases when we want to evolve a particular architecture, this set can have very different functions than the ones in the function set.

2.2.2 – Establishing Connections

Lets first look at what happens when any program associated with some node includes in its code one of the above terminals. The value of a network's input or of other node's output becomes available to some computation being done in this program, so we can say that a **connection** has been established between the node associated with the program an the input or node corresponding to the included terminal. Depending on the terminal we can have two types of connections:

1. **Feedforward connections** – when a i_j or a_j, terminal is included in a program, this program will have access to the values of the correspondent network's inputs and node's output in the same iteration *t* the program is run. This creates connections similar to the ones in feedforward neural networks. In our approach this connections are allowed from network's inputs to any node and from internal nodes to external nodes. This is guaranteed by the different terminal sets available to the programs associated with external and internal nodes. The main importance of feedforward connections in GPN is that they allow the establishment of **functional** relations between nodes, i.e. internal nodes can perform some computations obtaining intermediate values that external nodes can then use in more complex computations.
2. **Recurrent connections –** when a b_j terminal is included in a program, it will have access to the output value of the node correspondent to that terminal in the iteration prior to the one in which the program is being run. This way, a **recurrent** connection is established from the node associated with the terminal to the node associated with the program that includes the terminal. These connections are similar to the ones allowed in recurrent neural networks or in cellular automata and they can exist between any two nodes in a network. Their extreme importance in GPN comes from the fact that they allow **temporal** relations to be established between nodes. This means that a sequence of *n* recurrent connections between nodes can be used in a GPN to keep available a value computed *n* iterations ago. The evolutionary process creates recurrent connections between nodes to implement memory needed to keep track of previous system's states.

The process of establishing connections in a GPN gives us the first difference between internal and external nodes: forward connections can only be established from internal nodes to external ones. It also explains two of the most interesting features of GPN: the easy way of creating temporal and functional relations between nodes and the implicit way this relations are created, which frees the evolutionary process of maintaining any explicit representation for connections between nodes.

2.2.3 – Running the GPN

Programs evolved using genetic programming are the processing elements in a GPN. To run the GPN all the programs associated with the network's nodes must also be run. In an ideal parallel implementation of a GPN all programs, or, at least, subsets of programs, could be run simultaneously. Like most connectionist structures it seems that the most natural implementation for a GPN would be a distributed one. In a sequential implementation, internal nodes are the first ones to be run, so that their outputs are available to propagation by forward connections to external nodes. External nodes are then run sequentially and their output copied to the network's outputs. This procedure emphasises the other major difference between internal and external nodes: each external node has an associated network's output and the output value of the i^{th} external node at iteration t corresponds to the value of the i^{th} network's output. This implies that the number of network's outputs and external nodes is always the same.

2.3 – The Environment

The environment presents the agent with problems the GPN must lead the agent to solve. Sensorial information obtained by the agent from the environment is fed to the GPN and the agent executes the resulting output command. The sequence of executed commands results in an attempt by the agent to solve the proposed problem. The environment must supply a procedure capable of computing a quantitative measure of the agent fitness, or ability to solve the problem. Like in all evolutionary techniques our system relies on this fitness measure to drive the search for a potential solution.

3 – Evolving Genetically Programmed Networks

To successfully evolve a GPN, we must devise a representation scheme for the individuals, as well as the operators that will act on them.

A GPN individual has ***n* chromosomes**, one for each node in the network. Since the chromosomes correspond to the programmes associated with the networks' nodes, it follows that each individual I_i (a GPN) is simply represented by a sequence of programs:

$$I_i=\{P_1, P_2, \ldots, P_n\},$$

with P_n the program associated with the network's n^{th} node. Order is important, since, like in the biological model, the genome operators are designed to act on chromosomes with the same *position* in the individuals they belong to.

3.1 – Operators

The three main operators used in GP are also needed to evolve GPNs, therefore we must define reproduction, crossover and mutation in a way they can be applied not only to a program but also to a GPN, a sequence of programs. What we did was, basically, to define the three operators at two different levels: at individual level and at program level. At individual (GPN) level we than have:

- **Reproduction** – reproduction is an asexual operator which, given a certain individual returns an exact copy as its child. To achieve this, program reproduction must be applied to every program associated with the individual's nodes. The resulting programs will be associated with the correspondent nodes in the new individual.
- **Crossover** – crossover is a sexual operator which, given two individuals, produces two children by recombination of the original individuals. Applying crossover to two individuals implies applying program crossover to every pair of correspondent programs in these individuals.
- **Mutation** – mutation introduces random changes into an individual. To produce a mutated child of a given individual program, mutation is applied to every program in the parent before it is copied to the correspondent node in the child.

At program level operators are defined in a way similar to GP:

- **Program Reproduction** – the program reproduction operator returns an exact copy of the parent program as the child program.
- **Program Crossover** – The program crossover operator takes two programs and returns two children resulting from the recombination of the parents. To do this, a sub-tree is identified in each of the parents, with its root node randomly chosen. These sub-trees will then be swapped in the new programs. When crossover is not possible (e.g. one of the children would be larger than the maximum size allowed) copies of the two parents are returned as the children, and crossover degenerates into program reproduction.
- **Program Mutation** – A mutated child program is obtained by substituting a randomly chosen sub-tree of the program for a new, randomly generated, sub-tree.

3.2 – The Initial Population

An initial population composed of n individuals with m nodes will imply the generation of $n*m$ programs. Since that at this stage in our study of GPNs we don't allow the individuals' number of nodes to vary, neither inside a population nor during the evo-

lutionary process, this number will remain constant during the evolutionary process. The generation of this initial set of programs is done using what [Koza92] calls the "grow" method: a restriction is made on the tree's maximum depth and, while this depth is not reached, nodes are randomly chosen from the reunion of function and terminal sets. When the maximum depth is reached nodes are randomly chosen from the terminal set alone, which causes the tree's depth to be no greater then the maximum depth allowed. This method creates trees of different sizes and shapes and revealed itself appropriate to be used with our approach.

3.3 – Selection and Evolution

Tournament selection, usually of size n=4, was used in all of our experiments. Candidates for reproduction are chosen after entering a tournament between n individuals randomly chosen from the current population. The individual with the best raw fitness is considered to be the winner and is copied into a mating pool with the same size as the population. After the mating pool is full, reproduction is applied to 10% of the individuals, mutation to 5% of the individuals and crossover to the remaining ones. The individuals resulting from the operator application are copied into a new population. The process ends at generation 50 (the initial population generation is considered to be generation 0) or when an individual with some goal fitness is found.

4 - Experimental Results

4.1 – The Ant Problem

As a first test to the method described in the previous sections we needed a relatively simple control problem, extensively studied in literature and preferably having been the object of approaches using evolutionary techniques. A problem that fulfils these requirements is the usually called Ant Problem. Originally presented in [Collins91], this problem consists in developing a controller capable of guiding an artificial ant in a toroidal 32×32 cell world so that the ant correctly follows a discontinuous trail of sugar. The ant has a rudimentary sensor, which informs it if there is sugar in next cell in the direction of its movement. The ant can perform four actions: turning left or right while remaining in the same cell, moving one cell in the current direction or doing nothing. The total number of actions the ant can perform to follow the complete trail is usually limited.

We can find many evolutionary approaches to this problem. [Collins91] uses a genetic algorithm (GA) to evolve both neural networks and finite automata capable of following the "John Muir" trail, [Angeline93] and [Pujol98] use, respectively, a GA and Parallel Distributed Genetic Programming (PDGP) to evolve neural networks capable of following the same trail. The variant of this problem we will try to solve is called the "Santa Fé" trail and it has been extensively used as a benchmark problem for GP. It was first presented in [Koza92], and is harder, with more levels of deception than the "John Muir" trail. [Langdon98] presents an extensive study of the program space for this problem in GP, and compares the effort needed to find a solution with random search, GP and several other search techniques. Effort is defined as in [Koza92] to be number of individuals that need to be evaluated to ensure a solution is found, with probability z, and can be computed using the following equations:

$$R(m,i,z) = ceil\left(\frac{\log(1-z)}{\log(1-P(m,i))}\right)$$

$$P(m,i) = \frac{S(i)}{n}$$

$$I(m,i,z) = m \times R(m,i,z) \times (i+1)$$

with $P(m,i)$ the cumulative probability of success at iteration i with a population m; $S(i)$ the number of successful runs at iteration i, n the total number of runs; $R(m,i,z)$ the number of runs needed to find a solution at iteration i, with a population m and probability z; and $I(m,i,z)$ the number of individuals that must be evaluated to guarantee that a solution is found with probability z. Table 1 presents, for comparative purposes, values of $I(m,i,z)$ for several approaches. Most of these values were taken from [Langdon98]. The value for evolutionary programming (EP) was taken from [Chellapilla97].

Method	***I(m,i,z)***
GP	450,000
Sub-Tree Mutation	426,000
PDGP	336,000
Strict Hill Climbing	186,000
EP	136,000

Table 1: Comparative results over the number of individuals that must be evaluated to find a solution with probability 0.99 for the "Santa Fe" trail.

From Table 1 we can see that the best results are those obtained by [Chellapilla97]. These are obtained using a form of evolutionary programming with three mutation operators to evolve programs capable of solving "Santa Fe" trail problem. In [Langdon98a] better results are obtained, but the problem is changed: the ant is only allowed to see the next n sugar cubes, and the next n are only inserted when the previous n are collected.

4.2 – Using GPN to Solve the Ant Problem

In this section we will present three different approaches to the described problem by using GPNs to

evolve a distributed program, a rule-based system and a neural network, all capable of leading the ant trough the "Santa Fe" trail. Table 2 presents the parameters that are common to all experiments in this section. Most of them are similar to ones used in previous approaches. We use a small population of only 100 individuals to make viable, given the computational needs of our system and the resources available to us, the performance of a substantial number of runs.

Objective	Control an ant so that it eats all 89 pieces of sugar on the "Santa Fe" trail
Fitness	Sugar eaten after 400 commands
Population size	100
Number of gener.	51
Number of runs	200
Selection method	Size 4 tournament
Crossover Prob.	85%
Reproduction Prob.	10%
Mutation Prob.	5%

Table 2: Parameters common to all the experiments on this section.

4.2.1 – Evolving a Distributed Program

In our first attempt to use a GPN to solve the Ant Problem will try to evolve what we call a distributed program. This is the simplest way of using GPN, since it doesn't involve any particular restriction to induce the evolution of some desired architecture. Our first step in solving the problem will be the definition of the networks inputs and outputs. We chose to use 2 inputs with the values 10 when there is sugar in front of the ant and 01 when there is not. We use three outputs, each one corresponding to an available action: **move**, **turn right**, **turn left** (like in most GP based approaches **doing nothing** is not used). The action performed should correspond to the output with larger value. The next step concerns the decision on the number of internal and external nodes. As already described there must be an external node for each output, sob there are three external nodes. Since our method does not yet allow the number of internal nodes to be evolved, we also have to decide on it. After some empirical study we chose to use 6 internal nodes. These parameters concerning GPN topology were kept constant over all experiments and are summarised in Table 3.

Number of Inputs	2
Number of Outputs	3
External Nodes	3
Internal Nodes	6

Table 3: GPN parameters kept constant over all experiments.

To allow the evolution of the GPN we must now define the function set F, the root set R and the terminal sets T_i and T_e. What we want the GPN to involve into is a program, distributed over the several nodes of the network as smaller subprograms. These programs must have access to each other output values and to the networks inputs so they can, working co-operatively, compute a valid output in response to the present inputs and the system state. An important feature of the GPN is the way the system state is kept. Much like recurrent neural networks, GPN implement a memory of the system current state by allowing the establishment of recurrent connections between nodes, i.e. programs can access the outputs of other programs in the previous iteration. To allow the subprograms to access input values and other programs' output in the previous iteration, we must include in the terminal set, terminals corresponding to this data. We could also allow feedforward connections, i.e. allow the external nodes' programs to access the output of the internal nodes' programs in the current iteration. Since this doesn't seem necessary to solve our problem, will keep the correspondent terminals out of terminal set by now. This implies that the terminal set for internal (T_i) and external (T_e) nodes will be the same. The function set must have the functions needed to compute the output values. Since binary output values are enough for the problem at hand, we will only include logical, comparative and an *if* function in the function set. Finally, the root set, which correspond to functions that can be the root of the program tree. Since we don't want to make any particular restrictions to the tree's root, the root set will have the same elements as the function set. Terminal, function and root set for evolving a distributed program capable of leading the ant through the "Santa Fe" trail are presented in Table 4.

$T_i=\{i_1, i_2, a_1, a_2, \ldots, a_9\}$
$T_e=\{i_1, i_2, a_1, a_2, \ldots, a_9\}$
$F=\{and, or, not, >, <, ==, !=, if\}$
$R=\{and, or, not, >, <, ==, !=, if\}$

Table 4: Terminal, function and root sets for evolving a distributed program capable of solving the Ant Problem.

It is important to note that none of this terminals or functions are problem dependent or have any kind of secondary effect, as is usual in GP. Using the above settings 200 independent runs were carried out and in each run a population of 100 genetically pro-

grammed networks was evolved for 51 generations. The obtained results are presented in **Figure 3**. The topology of a successful distributed program is presented in **Figure 4**. **Figure 5** shows a sample program evolved for a node in this solution.

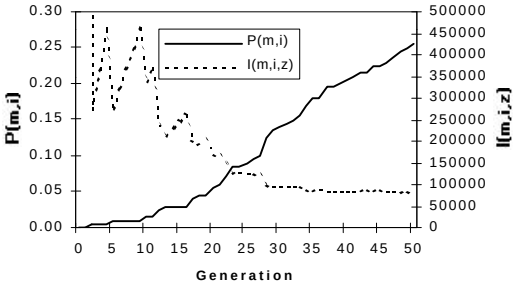


Figure 3: Results for evolving a GPN as a distributed program. The cumulative probability of success, *P(m,i)*, is *0.255* at generation *50*, and the smaller number of individuals that need to be processed, *I(m,i,z)*, with *z=0.99* is *81,600* achieved at iteration *50*.

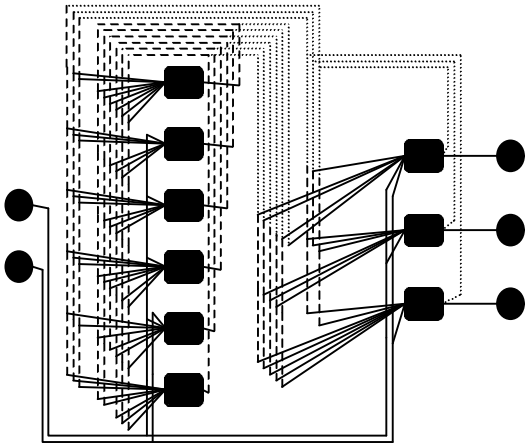


Figure 4: Topology of a GPN solution for the ant problem: a distributed program. There are no forward connections from internal to external nodes.

```
(or agent[6] (and (< (if (== agent[8] agent[6]) agent[0] agent[2])
  (not (if (< agent[6] (!= (or agent[6] (and (not agent[6])
  (not agent[4]))) (> (if agent[1] (!= (not agent[6]) agent[1])
  (== agent[1] agent[6])) agent[4]))) agent[3] agent[3])))
  (not (< (!= (or agent[1] agent[6]) agent[2]) agent[7]))))
```

Figure 5: Evolved program for a node in a distributed program solution for the ant problem. Agent terminals imply recurrent connections to other nodes.

The results obtained with this approach are substantially better, in terms of the number of individuals that must be processed so that a solution is found with 99% probability, than the ones obtained using other methods. From Table 1 we can see that "standard" GP needed to evaluate 450,000 individuals, while [Chellapilla97], using EP, has the best results known to us, with 136,000 individuals. Using genetic programming to evolve genetically programmed networks as distributed programs we need only to evaluate 81,600 individuals.

4.2.2 – Evolving a Rule Based System

To fulfil our initial goal of allowing GPNs to be evolved to different architectures we will evolve a rule-based system capable of solving the Ant Problem. This system will be composed by a set of rules of the form:

if <*condition*> **then** <*conclusion1*>
else <*conclusion2*>

The evolution of such a system is simply done by applying some changes to the terminal, function and root sets used in the previous point to evolve a distributed program. The main change is the removal of the *if* function from function set while the root set is modified to have only the same *if* function. This will produce programs with the structure of the above rule, where *condition*, *conclusion1* and *conclusion2* will be subprograms composed of the instructions in the function set. The *knowledge* produced by a rule is the result of the subprogram executed as the rule conclusion. To allow this *knowledge* to be used by other rules in the same iteration, forward connections are allowed from internal nodes to external ones, and the T_e set is changed accordingly. Recurrent connections are also allowed so the rules can use the *knowledge* produced by other rules in the previous iteration. Terminal, function and root sets used in this experience are presented in Table 5.

$T_i=\{i_1, i_2, a_1, a_2, \ldots, a_9\}$
$T_e=\{i_1, i_2, a_1, a_2, \ldots, a_9, b_1, b_2, \ldots, b_6\}$
$F=\{and, or, not, >, <, ==, !=\}$
$R=\{if\}$

Table 5: Terminal, function and root sets for evolving a rule-based system capable of solving the Ant Problem

For the same 200 runs, we obtained the results presented in **Figure 6**. The topology of a successful rule based system is presented in **Figure 7**. **Figure 8** shows a sample program evolved for an external node in this solution. The results are still better than the ones of all other non-GPN approaches but don't improve on the ones obtained when evolving the distributed program based solution.

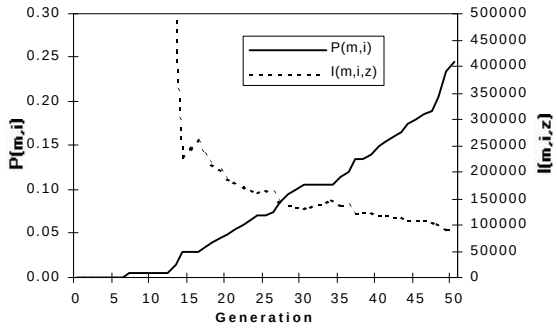


Figure 6: Results for evolving a GPN as a rule based system. The cumulative probability of success, *P(m,i)*, is *0.245* at generation *50*, and the smaller number of individuals that need to be processed, *I(m,i,z)*, with *z=0.99* is *86,700* achieved at iteration *50*.

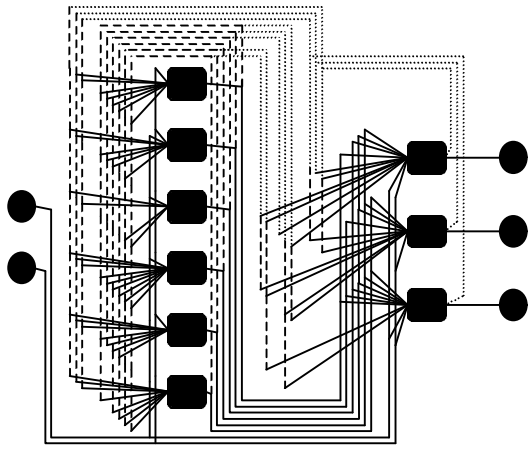


Figure 7: Topology of a GPN solution for the ant problem: a rule based system. Forward connections are allowed only from internal to external nodes.

```
(if ag[1] (and (and (> (and in[0] (not ag[4])) (or (< in[1] dag[1])
   (not (< ag[3] (!= (== in[0] ag[2]) (!= (== (and ag[5] in[1])
   ag[4]) (and in[1] dag[4])))))) ag[0]) dag[4]) in[1])
```

Figure 8: Evolved program for an external node in rule based solution for the ant problem. Ag and dag terminals imply, respectively, forward and recurrent connections to other nodes.

4.2.3 – Evolving a Recurrent Neural Network

To evolve a recurrent neural network, the changes we have to make to the terminal, function and root sets are more substantial. The most important task is that of finding a way to evolve the connection's weights, since the way of evolving the connections themselves is inherent to our approach. The method we use to evolve the weights in connections is based in the attachment of a random number between 0 and 2.5 to every terminal appearing in a program. As the only functions in the function set are *add* and *subtract*, each program its no more than a linear combination of the node's available inputs multiplied by different values each time they appear. This means that for each terminal the evolutionary process will try to find a correct combination of values so that a correct weight for the correspondent connection is produced. We already mentioned that the function set has only two members: *add* and *subtract*, but we haven't said anything about the root set. This has only one element, the transference function *transf*. *Transf* has only one argument and returns this argument if its value is between 0 and 1, returns 0 if the value is less or equal to 0, and returns 1 if the value is more than 1. We could also have used several transference functions, leaving to the evolutionary process the selection of the most appropriate ones. Terminal, function and root sets used in this experience are presented in Table 6. The *w* before the terminals means they have an associated weight.

T_i={wi_1, wi_2, wa_1, wa_2, …, wa_9}
T_e={wi_1, wi_2,wa_1,wa_2, …, wa_9, wb_1, wb_2, …, wb_6}
F={+, -}
R={*transf*}

Table 6: Terminal, function and root sets for evolving a recurrent neural network capable of solving the Ant Problem

For the same 200 runs, we obtained the results presented in **Figure 9**. The topology of a successful neural network is presented in **Figure 10**. **Figure 11** shows a sample program evolved for an external node in this solution. These were the best results obtained by us, only needing to evaluate 59,500 individuals to find, with a probability of 99% a neural network capable of correctly guiding the ant to collect all the 89 sugar cubes on the "Santa Fe" trail.

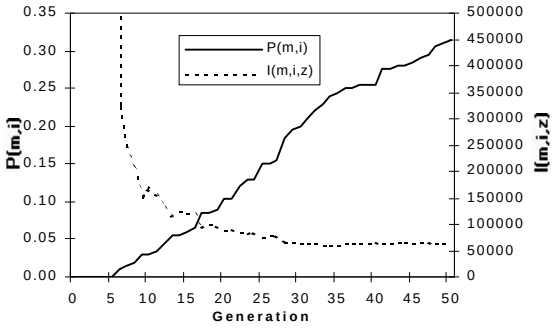


Figure 9: Results for evolving a GPN as a recurrent neural network. The cumulative probability of success, *P(m,i)*, is *0.315* at generation *50*, and the smaller number of individuals that need to be processed, *I(m,i,z)*, with *z=0.99* is 59,*500* achieved at iteration *34*.

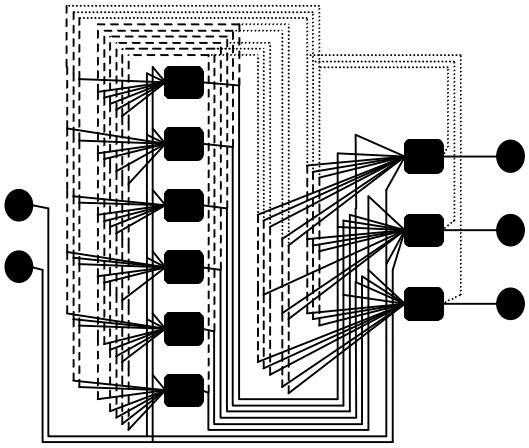


Figure 10: Topology of a GPN solution for the ant problem: a recurrent neural network. Forward connections are allowed only from internal to external nodes.

```
(transf (+ (+ (+ (+ (- (+ 0.57*dagent[5] (+ (- (+ (+ (+
  (+ (+ (+ 2.32*dagent[4] 1.49*dagent[2])
  (+ 1.49*dagent[2] 0.46*dagent[3])) 1.25*in[1])
  (+ (- (- (- 0.87*in[1] 0.80*dagent[0]) 0.32*dagent[8])
  0.84*dagent[3]) 0.88*dagent[3])) 0.57*dagent[5])
  0.82*in[1]) 1.49*dagent[2]) 1.96*dagent[7]))
  1.93*in[1]) (+ 1.78*in[1] 1.49*dagent[2]))
  1.49*dagent[2]) 1.78*in[1]) 1.12*dagent[7]))
```

Figure 11: Evolved program for an internal node in neural network based solution for the ant problem. Agent and dagent terminals imply, respectively, forward and recurrent connections to other nodes.

Results obtained in the three GPN based approaches are summarised in Table 7, together with the better non-GPN results.

Method	***I(m,i,z)***
EP	136,000
GPN (Dist. Program)	81,600
GPN (Rule Based System)	86,700
GPN (Neural Network)	59,500

Table 7: Comparative results over three GPN based approaches and the best known non-GPN approach for the "Santa Fe" trail problem.

5 – Conclusions

In this article we described a new method to generate several forms of distributed programs, and applied it to the evolution of controllers for simple agents. This method is based on a connectionist structure, which we call Genetically Programmed Network, where each node has an associated program, which is evolved using genetic programming. Each GPN individual has several nodes, so its genome is a sequence of chromosomes, each one corresponding to a program. Manipulating the function, terminal and root set of the programs, we showed that it was possible to evolve GPNs into controllers with very different architectures.

As an example, we applied the described method to a commonly used benchmark problem, the "Santa Fe" trail problem, and evolved populations of GPNs into distributed programs, rule based systems and neural networks all capable of solving the given problem. By comparing our results to the ones of several other approaches to this problem, we concluded that the number of individuals we need to be evaluated so that a solution is found, with a probability of 0.99, was substantially less using GPN than in the best of those approaches. We believe this can be justified by the highly connectionist nature of our approach, which allows that both functional and temporal relations between nodes can be easily created by forward and recurrent connections. This implies that memory mechanisms needed to solve problems where the previous state of the system must be remembered could be easily implemented by recurrent (delayed) connections between nodes. Forward connections are expected to allow the straightforward evolution of communities of small programs with distributed functionality, which should be easier to evolve than a larger, isolated, program capable of providing the same functionality.

We must finally emphasise that the plasticity of genetically programmed networks could be useful in bring us some insight into which architecture would perform better or be easily evolved for a given problem. New or hybrid architectures should also be easy to evolve and investigate. We also hope that by choosing a distributed and connectionist structure as the basis of this approach, and by evolving it using genetic programming, another of bridge can be made between connectionist and GP approaches.

6 - Ongoing and Future Work

In this work we have chosen to explore the polymorphic possibilities of the GPN, evolving several solutions with different architectures for the same problem. To prove this method is of real interest we must apply it to a range of other problems, and we must be able to evolve yet more different control architectures. This is part of the ongoing work on GPN. Another important point concerns the current limitation of our system to a fixed number of internal and external nodes. We hope to deal with this limitation in the near future.

References

[Angeline93] P. Angeline, "**Evolutionary Algorithms and Emergent Intelligence**", PhD Thesis, Ohio State University, 1993.

[Bhanzaf97] 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.

[Brooks92] 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.

[Chellapilla97] 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.

[Cliff92] D. Cliff, P. Husbands and I. Harvey, "**Analysis of Evolved Sensory-Motor Controllers**", *Cognitive Science Research Paper*, Serial Nº CSRP 264, 1992.

[Collins91] 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.

[Donnart92] 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.

[Dorigo93] 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.

[Dorigo93a] M. Dorigo and M. Colombetti, "**Robot Shaping: Developing Autonomous Agents through Learning**", *Technical Report TR-92-040*, International Computer Science Institute, April 1993

[Floreano94] 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.

[Grefenstette94] 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.

[Harvey93] 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.

[Keith90] M. Keith, "**Genetic Programming in C++: Implementation Issues**", *Advances in Genetic Programming*, pp. 285-310, MIT Press, 1994.

[Koza92] J. Koza, "**Genetic programming: On the programming of computers by means of natural selection"**, Cambridge, MA, MIT Press, 1992.

[Langdon98] W. Langdon and R. Poli, "**Why Ants are Hard**", Technical Report CSRP-98-04, The University of Birmingham, School of Computer Science, 1998.

[Langdon98a] W. Langdon, "**Better Trained Ants**", Technical Report CSRP-98-08, The University of Birmingham, School of Computer Science, 1998.

[Nolfi94] 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.

[Pujol98] J. Pujol and R.Poli, "**Efficient Evolution of Asymmetric Recurrent Neural Networks Using a PDGP-inspired Two-dimensional Representation**", *Proceedings of EuroGP'98, First European Workshop on Genetic Programming*, April 1998, Springer.