Building Agents with Memory: An Approach using Genetically Programmed Networks

Arlindo Silva[1]
Center for Informatic and Systems of the University of Coimbra
Polo II – Pinhal de Marrocos
3030 Coimbra - Portugal
arlindo@dei.uc.pt

Ana Neves[1]
Center for Informatic and Systems of the University of Coimbra
Polo II – Pinhal de Marrocos
3030 Coimbra - Portugal
dorian@dei.uc.pt

Ernesto Costa[2]
Center for Informatic and Systems of the University of Coimbra
Polo II – Pinhal de Marrocos
3030 Coimbra - Portugal
ernesto@dei.uc.pt

Abstract- To achieve a high degree of autonomy, an agent usually needs some kind of memory mechanism. In this article we present a new approach to the evolution of agents with memory, based on the use of Genetically Programmed Networks. These are connectionist structures where each node has an associated program, evolved using genetic programming. Genetically Programmed Networks can easily be evolved into agents with very different architectures. We present experimental results from evolving Genetically Programmed Networks as neural networks, distributed programs and rule-based systems capable of solving problems where the use of memory by the agent is essential. Comparisons are made between the performance of these solutions and the performance of solutions obtained by other evolutionary strategies used to evolve agents with memory

1 Introduction

Autonomous agents need memory to encode the state information needed to efficiently solve complex problems. In most of the approaches where some form of genetic programming is used to evolve agents capable of solving problems where memory is essential, memory is implicitly implemented by the program structure. The state of the agent is represented not by some explicit memory construct but as the emergent result of executing a sequence of instructions [Koza92]. Explicit memory can be implemented using indexed memory [Teller94]. The agent's state is stored in a sequence of memory cells, which the evolved program can read and write. [Trenaman98] proposes a form of concurrent genetic programming where state information can be encoded by the exchanging of processor control between threads executed concurrently. Each thread corresponds to a genetically evolved program. [Angeline97] Multiple Interacting Programs (MIP) implement memory in a way similar to Genetically Programmed Networks (GPN): a MIP is a set of equations evolved using evolutionary programming (10 different forms of mutation). Each equation is associated with a node of a network (which can be seen as a generalisation of neural networks) and memory is implemented by creating recurrent connections between nodes.

In this article we propose Genetically Programmed Networks as an alternative process to evolve agents with memory. Based on results obtained using GPN to solve two benchmark problems, the Ant Problem and the Tartarus Problem, we defend that evolving GPN can be more efficient than other evolutionary approaches previously used to evolve agents with memory. We also demonstrate that a GPN is a polymorphic structure, by evolving solutions with very different architectures for the same problem. More specifically, we use GPN to evolve distributed programs, rule based systems and neural networks all capable of solving the problems mentioned.

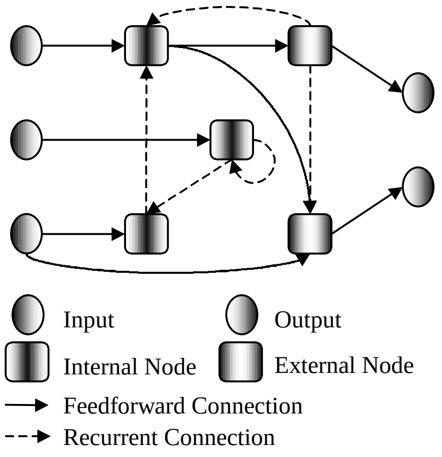

Figure 1: A Genetically Programmed Network

[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

A Genetically Programmed Network is constituted by a sequence of programs. Each program can be pictured as associated with a node in a network. Besides the nodes with associated programs, the network also has a set of inputs, a set of outputs and connections between them (see Figure1). 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 inputs, outputs and nodes. There is no explicit representation for this network. In fact, the network structure is implicitly defined in the way the programs make use of a carefully defined terminal set. The evolutionary process used to evolve the programs, and, consequently, the GPN, is an extension of genetic programming (GP), as defined in [Koza92], to multi-tree individuals.

Genetic programmed networks are described in section 2 and the evolutionary process used to evolve valid solutions from a GPN population is explained in section 3. In section 4 we present the experimental results obtained by evolving GPN as distributed programs, recurrent neural networks and rule-based systems, all capable of solving the benchmark problems. In the same section we compare the efficiency of our approach with other methods of evolving agents with memory capable of solving the same problems. We discuss how GPN can be used to evolve agents with different architectures in section 5 and, in section 6, we draw some conclusions about this results and outline ongoing and future work.

2 Genetically Programmed Networks

Genetically Programmed Networks are sequences of programs, which implicitly define a connectionist structure. This structure, a network of nodes with attached, genetically evolved programs, gives the name to our approach. In this section we briefly describe the structure and behaviour of a GPN. For a more detailed description see [Silva99].

2.1 GPN General Structure

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

- A set of inputs, whose values are received from the environment.
- A set of outputs, whose values are computed by the GPN.
- 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:

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

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*.

2.2 GPN Behaviour

The desired behaviour for a GPN is to solve some problem posed by the environment. To do this, the network uses the information provided by the environment at its inputs and produces outputs that can be seen as instructions to eventually achieve a solution. 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's 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. In GPN these sets are not as problem dependent as in GP, since no side effects are allowed. On the other hand, these sets are heavily dependent on architectural choices. In fact, we must 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. The kind of connections allowed will also depend on the chosen terminals.

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. A terminal set can have the following inputs: network's inputs at iteration *t*, nodes' outputs at instant *t* and nodes' outputs at instant *t-1*. By restricting the inputs (terminals) available to a given program we can restrict the connections the correspondent node can make, thus affecting the networks topology.

Instead of using only one function set we use two: a function set and a root set. The members of the root set are the functions that can be chosen to be the root node in the program's tree. Functions in the function set are used in the same way as in GP, except that they cannot be chosen to be the program's tree root node. The use of a root set allows us to control more easily the type of architecture being evolved.

2.2.2 Establishing Connections

Connections are established in a GPN when a program attached to some node includes in its code any terminal corresponding to one of the inputs available to the node. If a program, at iteration t, uses a terminal linked to a network's input or to a node's output at the same iteration *t*, we say that a forward connection has been established. Forward connections are only allowed between inputs and nodes, and between internal nodes and external nodes. These connections are useful to carry data between nodes, e.g. from an input to a node, and to define functional relations between nodes:

when there is a forward connection between nodes, the first node performs some computation which result the second node can use in its own computation.

When a program, at iteration *t*, uses a terminal linked to a node's output at instant *t-1*, a recurrent connection is established. Recurrent connections are allowed between any pair of nodes in the network. Temporal relations exist between nodes recurrently connected, since the second node will have access to the first node's output in the previous iteration. It is the existence of recurrent connections between nodes that is responsible for the implementation of memory in GPN. An agent can keep track of its state by using chains of recurrent connections to remember values obtained in some previous iteration. These values can than be modified by the following iterations in some way that reflects the changes in the agent state.

Defining the terminal set of a node as a subset of all the terminals allowed to that node, we can make restrictions on the topology of the GPN being evolved. E.g., we can allow recurrent connections to exist only between internal nodes. To achieve this, we define different terminal, function and root sets for internal and external nodes.

2.2.3 Running the GPN

A GPN is run by sequentially executing all the programs in it. The programs associated with internal nodes are the first to be executed. Then the programs associated with external nodes are executed. This sequence is the reason why forward connections between nodes can only exist from an internal node to an external node. Every external node is connected with a GPN output, which implies that we always have the same number of external nodes and outputs in a GPN. When a GPN is run, data flows from the inputs to the network's nodes, where it is processed by the associated programs. The nodes' outputs are then propagated trough forward and recurrent connections to other nodes. When all the external nodes' programs have been executed the external nodes' outputs are copied to the correspondent network's output and the GPN is ready for the next iteration.

3 Evolving Genetically Programmed Networks

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

- **GPN Reproduction** – Reproduction is an asexual operator, which returns an exact copy of the individual as the child. To apply reproduction to a GPN all programs in the parent individual are copied, in the same sequence, to the child.
- **GPN Crossover** – Crossover is a sexual operator, which takes two individuals and returns two children resulting from the recombination of the parents. In GPN crossover is implemented by exchanging sub-trees between all correspondent programs (with the same position) in the individuals.
- **GPN Mutation** – A mutated child GPN is obtained by substituting a randomly chosen sub-tree of each program in the individual 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 GPN we don't allow the individuals' number of nodes to vary, neither inside a population nor during the evolutionary 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, until this depth is reached, nodes are randomly chosen from the reunion of function and terminal sets, except for the root node, where a function from the root set must be chosen. 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, with size *n* depending on the set of experiments, was used as the selection strategy. 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 when a limit generation is reached or when an individual with some goal fitness is found.

4 Evolving Different Architectures

We have already mentioned the polymorphic ability of GPN. In this section we will describe the process that allows us to evolve a distributed program, a rule based system and a neural network from the same base structure, a GPN, only by adjusting the nodes' terminal, function and root sets. We will first define the following symbols: T_i, T_e, F_i, F_e, R_i and R_e represent, respectively, the terminal sets for internal and external nodes, the function sets for internal and external nodes and the root sets for internal and external nodes; i_p, is the terminal that contains the value of the network's *pth* input at iteration *t*, f_q is the terminal that contains the value of the output of the *qth* network's internal node at iteration *t* and r_r is the terminal that contains the value of the output of the *rth* network's node at iteration *t-1*.

4.1 Distributed programs

If we allow a population of GPN to evolve without any special constraints on terminal, function and root sets, the individuals will evolve as distributed programs. Table 1 presents the typical sets for evolving distributed programs. An internal node can have forward connections from the network's inputs and recurrent connections from every other node. External nodes can also have forward connections from internal nodes. The program associated with each node can also have access to other terminals needed for the problem being solved. Function and root sets have no particular constraints, they all have the same members: the functions needed to solve the problem.

$T_i=\{i_1, ..., i_p, r_1, \ldots, r_r, ...\}$
$T_e=\{i_1, ..., i_p, f_1, ..., f_q, r_1, \ldots, r_r, ...\}$
$F_i=F_e=R_i=R_e=\{...\}$

Table 1: Typical terminal, function and root sets for evolving a distributed program.

Evolving a GPN with these terminal, function and root sets, we will obtained a network of simple programs, each one doing some part of the computation needed to solve the problem. The GPN behaves as a distributed program.

4.2 Rule Based Systems

With a small change to the root sets we can evolve an agent with a more specific architecture, a rule based system (see Table 2). This is done by constraining the root sets to have just one function: *if-then-else*. Every node in the GPN will have the same structure, the one of a if-then-else rule. The GPN can now be though of as a rule system where each node correspond to a rule, and every rule can use the knowledge produced by other rules to produce its own knowledge, the node output.

$T_i=\{i_1, ..., i_p, r_1, \ldots, r_r, ...\}$
$T_e=\{i_1, ..., i_p, f_1, ..., f_q, r_1, \ldots, r_r, ...\}$
$R_i=R_e=\{\textit{if-then-else}\}$
$F_i=F_e=\{...\}$

Table 2: Typical terminal, function and root sets for evolving a rule based system.

4.3 Recurrent Neural Networks

To evolve a GPN as a recurrent neural network, further changes are needed to the terminal, function and root sets (see Table 3). Each terminal has an associated weight. These weights are randomly generated for each occurrence of the terminal in the programs of the initial population. The evolutionary process searches for a suitable linear combination of weighted terminals. To allow this, the function set has only two functions: + and -, while the root sets only have one argument, which is non-linear transference function. Each node acts as an artificial neurone, with its output equal to the result of applying the transference function to the linear combination of weighted inputs.

$T_i=\{wi_1, ..., wi_p, wr_1, \ldots, wr_r\}$
$T_e=\{ wf_1, ...,w f_q\}$
$R_i=R_e=\{transf\}$
$F_i=F_e=\{+,-\}$

Table 3: Typical terminal, function and root sets for evolving a recurrent neural network

For the problems in the next section we defined the terminal sets so that recurrent connections could only be made between internal nodes. External nodes can only have forward connections from internal nodes. It is easy to see that by manipulating the terminal sets a large number of other topologies could be produced. It could also be interesting to use several transference functions in the root sets and allow the evolutionary process to choose the best function, or combination of functions, for the problem being solved.

The same flexibility in terms of topology is also valid for distributed programs and rule-based systems. We chose to use the described topologies since they are very general and allow the evolutionary process to choose from a high number of possibilities the connections needed to solve the proposed problems.

5 Experimental Results

5.1 The Ant Problem

The first benchmark problem for which we present experimental results is the well-known Ant Problem. Originally presented in [Collins91], this problem consists in developing an artificial ant capable of following a discontinuous trail of sugar in a toroidal 32×32 cell world. 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. Memory is essential to solve the problem because the ant has a limited number of actions to follow the trail. The ant will have to remember the regularities in the trail so that the discontinuities don't take too many actions to overcome.

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] uses a GA to evolve neural networks capable of following the same trail. Finite automata and recurrent neural networks are an obvious solution to provide the memory the ant needs to solve the problem. The variant of this problem that we solved 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 4 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].

We used GPN to evolve distributed programs, rule based systems and recurrent neural networks, all capable of following the "Santa Fe" trail. In all experiments, we used GPN individuals with 2 inputs, 6 internal nodes and 3 external nodes (attached to 3 outputs). The external nodes corresponded, respectively, to the actions move, turn right and turn left (like in most GP based approaches doing nothing is not used). The output with larger value determines which action is to be executed. Table 5 presents the terminal, function and root sets used for the three sets of experiments.

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

Distributed Program
$T_i=\{i_1, ..., i_p, r_1, …, r_r\}$
$T_e=\{i_1, ..., i_p, f_1, ..., f_q, r_1, …, r_r\}$
$F_i=F_e=R_i=R_e=$ *{ and, or, not, >, <, ==, !=, if-then-else}*

Table 5: Terminal, function and root sets for evolving a distributed program capable of solving the Ant Problem.

Rule-Based System
$T_i=\{i_1, ..., i_p, r_1, …, r_r\}$
$T_e=\{i_1, ..., i_p, f_1, ..., f_q, r_1, …, r_r\}$
$R_i=R_e=$ *{if-then-else}*
$F_i=F_e=$ *{ and, or, not, >, <, ==, !=}*

Table 6: Terminal, function and root sets for evolving a rule based system capable of solving the Ant Problem.

Recurrent Neural Network
$T_i=\{wi_1, ..., wi_p, wr_1, …, wr_r\}$
$T_e=\{ wf_1, ...,w f_q\}$
$R_i=R_e=$ *{transf}*
$F_i=F_e=\{+,-\}$

Table 7: Terminal, function and root sets for evolving a recurrent neural network capable of solving the Ant Problem.

Using the sets in Table 5, Table 6 and Table 7, we performed 200 runs for each set of experiments, using a population of size 100. The evolution of the probability of success, *P(m,i)*, and of the smaller number of individuals that need to be processed, I(m,i,z), so that a solution has 99% probability of being found, is presented in Figure 2. We can see that we need to process **81,600** individuals to find a distributed program solution, **86,700** individuals to find a rule based system solution and **59,500** individuals to find neural network solution. Comparing these values with the ones for other approaches (see Table 4 and Table 8), GPN seem the most efficient in evolving an agent capable of solving the Ant Problem.

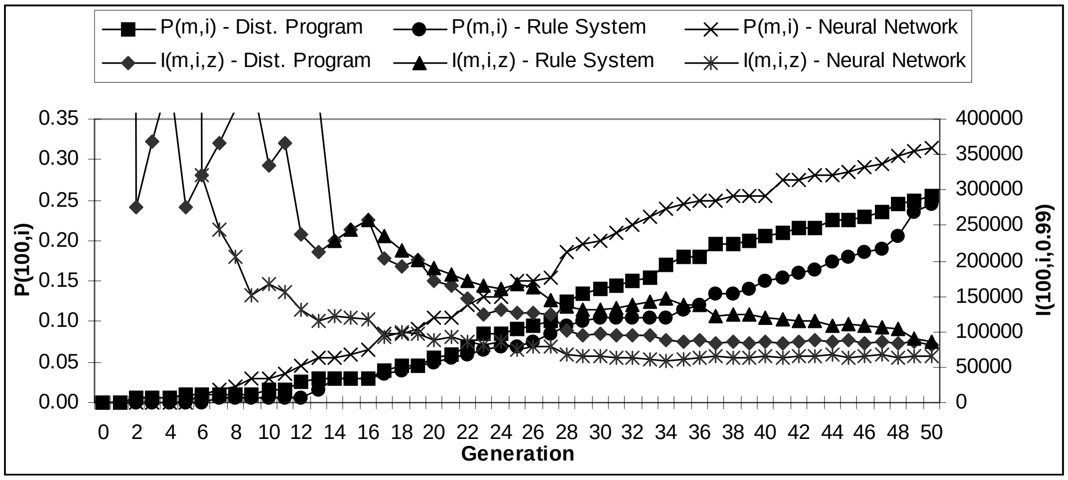

Figure 2: Evolution of the number of individuals that must be evaluated to find a solution with probability 0.99 for the "Santa Fe" trail, using three GPN based approaches.

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

Table 8: Comparative results over the number of individuals that must be evaluated to find a solution with probability 0.99 for the "Santa Fe" trail, using GPN based approaches.

5.2 The Tartarus Problem

The Tartarus Problem was first outlined in [Teller94] and is a good benchmark for evolving agents that need state memory. In our experiments we use a slightly different statement of the problem, as proposed in [Trenaman98]. Tartarus is a 6x6 bounded, cellular, world. In this version of the problem 5 blocks are randomly placed in the inner 4x4 cells. The agent's mission is to push the blocks into the world's boundaries in a fixed number of moves. The agent can move forward, turn right and turn left, and thus can move in eight directions. It also has 8 sensors, which return the state of the 8 neighbouring cells. A cell can be occupied, empty or a wall. The agent doesn't know its position or orientation and cannot push two blocks or a block into the wall. After 80 moves (move, right, left) the agent is awarded 2 point for each block in a corner and 1 point for each block in the edges. The fitness of the agent is the total score over 40 random worlds.

[Teller94] originally demonstrated that conventional GP agents perform poorly in this problem. [Teller94] incorporated indexed memory into its agents and showed that not only the agents performed better using memory but also that they performed badly if its memory was "damaged". These results proved that state memory is essential to agent performance in the Tartarus problem. [Trenaman] replicated these results and presented a new paradigm called Concurrent Genetic Programming (CGP), which implemented memory by using several scheduling strategies for two-tree agents which have their program trees executed concurrently. Table 9 summarises the results obtained by [Trenaman98] in terms of the best individual obtained in 100 generations and the average best individual over 25 independent runs. Results obtained using GPN based approaches are presented in Table 13.

Method	***Best***	***Average***
GP	98	63.2
GP + Ind. Memory	212	176.5
GP + ADF + Ind. Mem	207	165.1
CGP (best strategy)	221	205.0

Table 9: Comparative results for the best and average best individual for 100 generation runs.

GPN was used to evolve distributed programs, rule based systems and recurrent neural networks capable of performing well in the Tartarus Problem. Each GPN individual has 8 inputs (one for each sensor), 4 internal nodes and 3 external nodes (attached to 3 outputs). The external nodes corresponded, respectively, to the actions **move**, **turn right** and **turn left**. The output with larger value determines which action is to be executed. The terminal, function and root sets used for the three sets of experiments are presented in Table 10, Table 11 and Table 12. They are similar to the ones in Table 5, Table 6 and Table 7, used for the Ant Problem, except that the functions +, -, *, and / were added to the function sets for the distributed programs and ruled based system experiments.

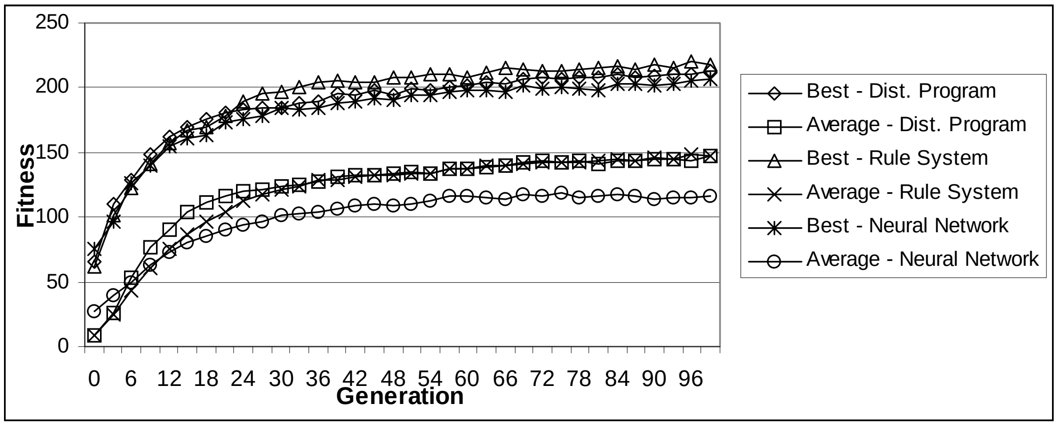

Figure 3: Evolution of the best individual and average fitness for three GPN based approaches to the Tartarus Problem

Distributed Program
T_i={i_1, ..., i_p, r_1, …, r_r}
T_e={i_1, ..., i_p, f_1, ..., f_q, r_1, …, r_r}
F_i=F_e=R_i=R_e={ and, or, not, >, <, ==, !=, +, -, /, if-then-else}

Table 10: Terminal, function and root sets for evolving a distributed program for the Tartarus Problem.

Method	***Best***	***Average***
GPN (Dist. Program)	264	218.9
GPN (Rule Based System)	252	226.7
GPN (Neural Network)	223	214.4

Table 13: Comparative results for the best and average best individual for 100 generation runs, for GPN based approaches.

Rule-Based System
T_i={i_1, ..., i_p, r_1, …, r_r}
T_e={i_1, ..., i_p, f_1, ..., f_q, r_1, …, r_r}
R_i=R_e={if-then-else}
*F_i=F_e={ and, or, not, >, <, ==, !=, +, -, *, /}*

Table 11: Terminal, function and root sets for evolving a rule based system for the Tartarus Problem.

Recurrent Neural Network
T_i={wi_1, ..., wi_p, wr_1, …, wr_r}
T_e={ wf_1, ...,w f_q}
R_i=R_e={transf}
F_i=F_e={+,-}

Table 12: Terminal, function and root sets for evolving a recurrent neural network for the Tartarus Problem.

We performed 20 runs for each set of experiments, using a population of size 800. The evolution over 100 generations of the best individual's fitness and the population's average fitness, averaged over the 20 runs, is presented in Figure 3. Table 5 shows the fitness for the best individual evolved over the 20 runs of 100 generations for each one of the architectures, and the average fitness of the best individuals in each run. Comparing with results for other approaches, GPN, especially when evolving the rule based system architecture, produces individuals with best fitness, not only over all runs, but also when averaging the best individuals evolved in each run.

6 Conclusions and Future Work

We presented results that seem to confirm our intuition that Genetically Programmed Networks can be a valid alternative for evolving agents with state memory. In the two benchmark problems used to test GPN, this approach behaved more efficiently that other GP based evolutionary approaches. It needed to evaluate less individuals to find a solution to the Ant problem, and found individuals fitter for the Tartarus Problem with the same generations limit and population size as the approaches used for comparison. While testing with other, more diverse, problems must still be done, these results seem very promising.

Another interesting point was the ability to evolve good solutions for the proposed problems using three different agent architectures: distributed programs, rule based systems and recurrent neural networks. While all of them were evolved from the same base connectionist architecture, the GPN, it interesting to see that some architectures seem fitter to produce solutions to a specific problem. E.g. the best results for the ant problem, with a meaningful advantage, were obtained evolving GPN as neural networks, while evolving rule based systems seem the most fit approach to the Tartarus Problem. Since changing the architecture being evolved is a very flexible and straightforward process, requiring only some changes the terminal, function and root sets of each group of nodes, GPN can be a useful tool in deciding where evolving a given architecture for a given problem is the most efficient approach.

Our current and future work involves the further study of the two aspects mentioned above. More experimental work must be done using new problems and evolving more diverse architectures.

7 Acknowledgements

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

Bibliography

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

[Angeline97] P. Angeline, "An alternative to indexed memory for evolving programs with explicit state representation" in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 431-438, Morgan Kaufmann, 13-16 July 1997.

[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.

[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.

[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.

[Silva99] A. Silva, A. Neves and E. Costa, "Evolving Controllers for Autonomous Agents Using Genetically Programmed Networks", accepted to be published in *EuroGP99: Proceedings of 2nd European Workshop on Genetic Programming.*

[Teller94] A. Teller, "The Evolution of Mental Models", *Advances in Genetic Programming*, Complex Adaptive Systems, pp. 199-220, MIT Press, 1994.

[Trenaman98] A. Trenaman, "Concurrent Genetic Programming and the Use of Explicit State to Evolve Agents in Partially-Known Environments", *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pp. 391-398, Morgan Kaufmann, 22-25 July 1998.