Evolving simple software agents: Comparing genetic algorithm and genetic programming performance

M.C. Sinclair and S.H. Shami*

Dept. of Electronic Systems Engineering, University of Essex,
Wivenhoe Park, Colchester, Essex CO4 3SQ.
Tel: 01206-872477; Fax: 01206-872900; Email: mcs@essex.ac.uk

Abstract

This paper investigates the relative efficiency of genetic algorithms and genetic programming in evolving simple software agents. The problem domain consists of an autonomous food-gathering agent placed on a square grid of hundred cells with food units spread evenly over the grid. Initial results show that evolving the agent using GP requires less effort than with GA. Nevertheless, further investigation revealed some interesting aspects.

1 Introduction

The aim of this paper is to compare the relative efficiency of genetic algorithms (GAs) [1] and genetic programming (GP) [2] for the evolution of simple software agents. This comparison forms part of a larger project to evolve software agents for distributed routing and restoration in telecommunications networks [3]. Consequently, within the overall timescales of the larger project, we were only able to devote a modest amount of effort to the work reported here. As the basis for the comparison, we independently re-implemented a simplified version of the recent work of Maskell & Wilby [4] (Model-1). They used a GA to evolve simple autonomous agents capable of collecting food on a square grid. We then created an equivalent system using GP to evolve agents for the same problem. It is our intention to use the insight gained from this necessarily limited comparison of these two approaches for evolving very simple agents to then guide us in our choice of approach for evolving the more complex agents that will be required in distributed network routing and restoration.

*Ph.D. student sponsored by the Government of Pakistan

2 Software Agents

Telecommunication networks are an area rich in research. With the ever increasing demand in volume and nature of services, network management has become more complex than ever. In the process, the vital functions of routing and restoration have seen a shift from centralised implementation to decentralised procedures [5]. Routing and restoration are multi-constraint optimisation problems that have to adapt and respond to a continuously changing network environment. Dynamic routing systems [6], as opposed to static ones, pro-actively update traffic routing by adapting in real-time to changes in network conditions. All these factors make it difficult for reliable distributed network control software to be developed by hand. This has led researchers to find alternative approaches to tackling the issue, including the use of agent-based software.

The theme behind intelligent software agents within networks is that they change the network management concept from complete control to partial control. In complete control a centralised controller dominates the network's behaviour all the time, whereas in partial control most jobs are delegated to the nodes. Essentially agents are software programs which, when placed in a given environment, interact with other agents as well as with that environment. A good agent has built-in *autonomy i.e.* it can work unaided; it is *reactive i.e.* it keeps interacting with the environment within appropriate time limits. Also agents are usually *pro-active i.e.* they can act on their own; and *social* in that they can communicate with other agents [7].

One promising approach in particular is evolving agent software using genetic techniques. Thus the

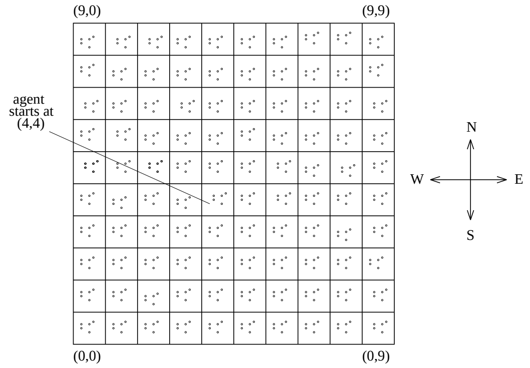


Figure 1: Grid

agents' high-level control code will be generated automatically and will hopefully thus be more robust (less 'brittle') than the hand-crafted equivalent. This is because evolved agent software does not seek to mimic human understanding of current network conditions, and thereby determine a course of action, but rather simply responds, with a high level of fitness, to the network's needs.

3 The Problem

However, before attempting to evolve the increasingly complex software agents required for distributed routing and restoration in telecommunications networks, we decided to spend a limited amount of time to first assess the relative efficiency of GAs and GP as the evolutionary mechanism. Given the earlier GA-based work of Maskell & Wilby [4], we decided to use this as the basis for comparison, despite the obvious simplicity of their agents, as it provided a ready-made benchmark aginst which to test GP.

In [4], Maskell & Wilby evolved a population of autonomous (*i.e.* non-communicative) food-gathering agents on a two-dimensional square grid of 100 cells, with a periodic boundary condition. In their problem (Model-1), each of the cells generated food at a low (barren), medium (normal) or high (oasis) rate. However, for our work, we chose the even simpler model of a uniform distribution of food on the grid (*i.e.* five units of food per cell). In addition, to reduce the potential noise in the fitness evaluation due to mutual interference of agents, we instead placed a single agent on a fully-restored grid for each evaluation (Fig. 1).

As in [4], the agent had access to certain high-level problem-specific functions (Table 1), with the invocation of a function taking one timestep. Each agent's fitness was taken to be the number of accumulated food units at the end of 25 timesteps.

Function	Action
Forage	Move either 1 or 2 cells depending on the first argument, collect (up to) either 1 or 2 food units depending on the second argument, and then rotate either left or right depending on the third argument.
Run	Move either 2, 4 or 6 cells depending on the first argument.
Pigout	Collect (up to) either 3 or 5 food units depending on the first argument.
Deposit	Drop (up to) either 2 or 4 units of food, depending on the first argument.
Wander	Move either 1 or 2 cells depending on the first argument, rotate either left or right depending on the second argument, and then move either 1 or 2 cells depending on the third argument.
Walk	Move 1 cell—no arguments used.
Nibble	Collect (up to) 1 food unit—no arguments used.
Nop	No action performed.
Jump	Set the instruction pointer to a different location within the program, depending on the first argument.
Turn	Rotate either 90, 180, 270 degrees depending on the first argument.

Table 1: Functions (after [4, TABLE 1])

At the start of a fitness evaluation, an agent was placed on a cell (*i.e.* `(4,4)`) facing `North`. Each timestep, the next instruction (function) was decoded (GA only) and invoked. Throughout the simulation, the agent would maintain a knowledge of its position (its `(x,y)` location), its direction (*i.e.* `North`, `South`, `East` or `West`) and the number of food units accumulated. It also had access to its program in binary string (GA) or tree (GP) form, with either an explicit (GA) or implicit (GP) instruction pointer (Fig. 2).

4 GA and GP Approaches

4.1 GA-based Agent

The program in a GA-based agent consisted of a concatenation of individual instructions. Close examination of Table 1 reveals that there are 39 distinct combinations of functions and arguments. Consequently, a 6-bit binary string was used to encode each instruction, with the $64 - 39 = 25$ redundant codes simply duplicating existing function and argument combinations. Given a ten instruction program length, this resulted in a 60-bit string.

A typical string at the end of a GA run is

```
110110111010110111010011110111010110110101010111110111100110
```

which would be decoded as shown in Table 2.

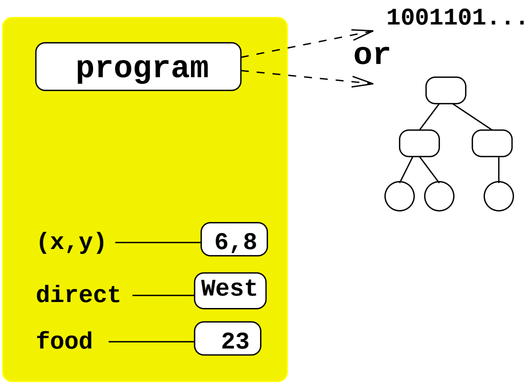


Figure 2: Agent data and program

String	Function
110110	(pigout 5)
111010	(walk)
110111	(pigout 5)
010011	(forage 1 1 Right)
110111	(pigout 5)
010110	(forage 1 2 Right)
110101	(pigout 3)
010111	(forage 1 2 Right)
110111	(pigout 5)
100110	(wander 2 Right 1)

Table 2: Decoded GA agent

4.2 GP-based Agent

Representing an agent's program as a parse tree was straightforward, except for: `jump`, which was omitted from the function set, some necessary modifications to the argument ranges, and the provision of return values to ensure closure [2].

The function set consisted of {`forage`, `run`, `pigout`, `deposit`, `wander`, `turn`} and the terminal set, {`walk`, `nibble`, `nop`, `0`, `1`, `2`}. Clearly, each of the values {`0`, `1`, `2`} had to be accepted by all of the function arguments; the argument interpretation and function return values are shown in Table 3.

A typical individual at the end of a GP run is

```
(forage (forage 1 1
                (pigout 2))
        (pigout 2) 1)
```

This s-expression converts into the tree shown in Fig. 3 and, with reference to Table 3, the tree would yield the following functions invocations:

```
(pigout 5)
(forage 1 1 2)
(pigout 5)
(forage 1 2 1)
```

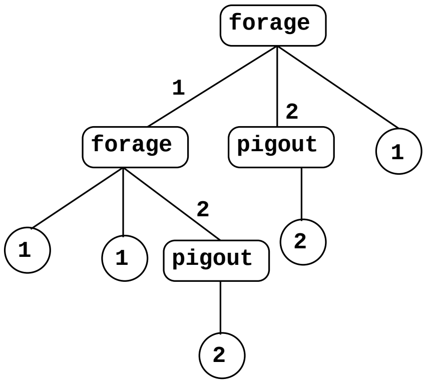


Figure 3: GP parse tree

4.3 Algorithm Details

To implement the GA approach, we used GENESIS v5.0 [9], with fitness-proportionate selection (with scaling), single-point crossover and mutation. The crossover probability used was 0.8, rather than the more usual 0.6, as it was found in trial runs to offer better performance. The large population size of 1000 was chosen for parity with GP. Further GA parameters used are given in Table 5. Our approach contrasted slightly with Maskell & Wilby [4] who used remainder stochastic sampling as the selection algorithm, two-point crossover, mutation, inversion and duplication for genetic operators.

We coded our agent in GP using `lil-gp` v1.02 [8]. The GP tableau is given in Table 4; other parameters were after Koza [2].

Input parameter	Value
Objective	Obtain an agent that collects as many food units as possible in 25 timesteps
Function set	{`forage`, `run`, `pigout`, `deposit`, `wander`, `turn`}
Terminal set	{`walk`, `nibble`, `nop`, `0`, `1`, `2`}
Fitness case	`(x,y) = (4,4)`; `direct = North`
Raw fitness	Number of food units collected
Standardised fitness	80 − Raw fitness
Hits	Raw fitness
Wrapper	None
Population size (M)	1000
Maximum generations (G)	31
Selection method	Fitness proportionate with overselection
Success predicate	None

Table 4: GP tableau

Function	Arg 1			Arg 2			Arg 3			Return
	0	1	2	0	1	2	0	1	2	
`forage`	0	1	2	0	1	2	`nop`	`Left`	`Right`	`arg2`
`run`	2	4	6							`arg1`
`pigout`	0	3	5							`arg1`
`deposit`	0	2	4							`arg1`
`wander`	0	1	2	0	1	2	0	1	2	`arg2`
`walk`										1
`nibble`										1
`nop`										0
`turn`	90	180	270							`arg1`

Table 3: Argument interpretation & return values

Input parameter	Value
Elitist	
Population size	1000
Maximum generations	31
Crossover rate	0.8
Mutation rate	0.001
Generation gap	1.0
Scaling window	5

Table 5: GA parameters

5 Initial Results

In our initial experiments 40 runs were carried out, first for the GA-based agent, and then for the GP-based agent. The population size (M) was fixed at 1000 and each run was allowed 31 generations ($i = 0$ to 30). The best raw fitness observed was 65 food units. This fitness would appear to be the actual optimum because, given a constraint of 25 timesteps, we have been unable to design one manually of greater fitness.

For GP it was observed that all 40 runs yielded the 65 food units target within 31 generations. However this was not the case for GA; this achieved 65 within 31 generations in 28 of the 40 runs. The remaining 12 runs were then extended to 200 generations. Four of these reached the optimum between Generation 32 and Generation 200. Eight stubborn runs did not achieve the target even in 200 generations, although for comparison purposes in the following calculations, we took the optimistic assumption that they did succeed in Generation 200.

Following Koza [2], from the 40 runs for each approach we first recorded the instantaneous probability of success for each generation i, denoted $Y(M,i)$, which in turn provided us with the cumulative probability of success, $P(M,i)$. We then computed the minimum number of multiple independent runs required ($R(z)$) to obtain an agent of raw fitness 65 by generation i, with a 99% probability. Finally for each generation we determined the number of individuals that would have to be processed in order to achieve 99% success, denoted $I(M,i,z) = M \times (i+1) \times R(z)$.

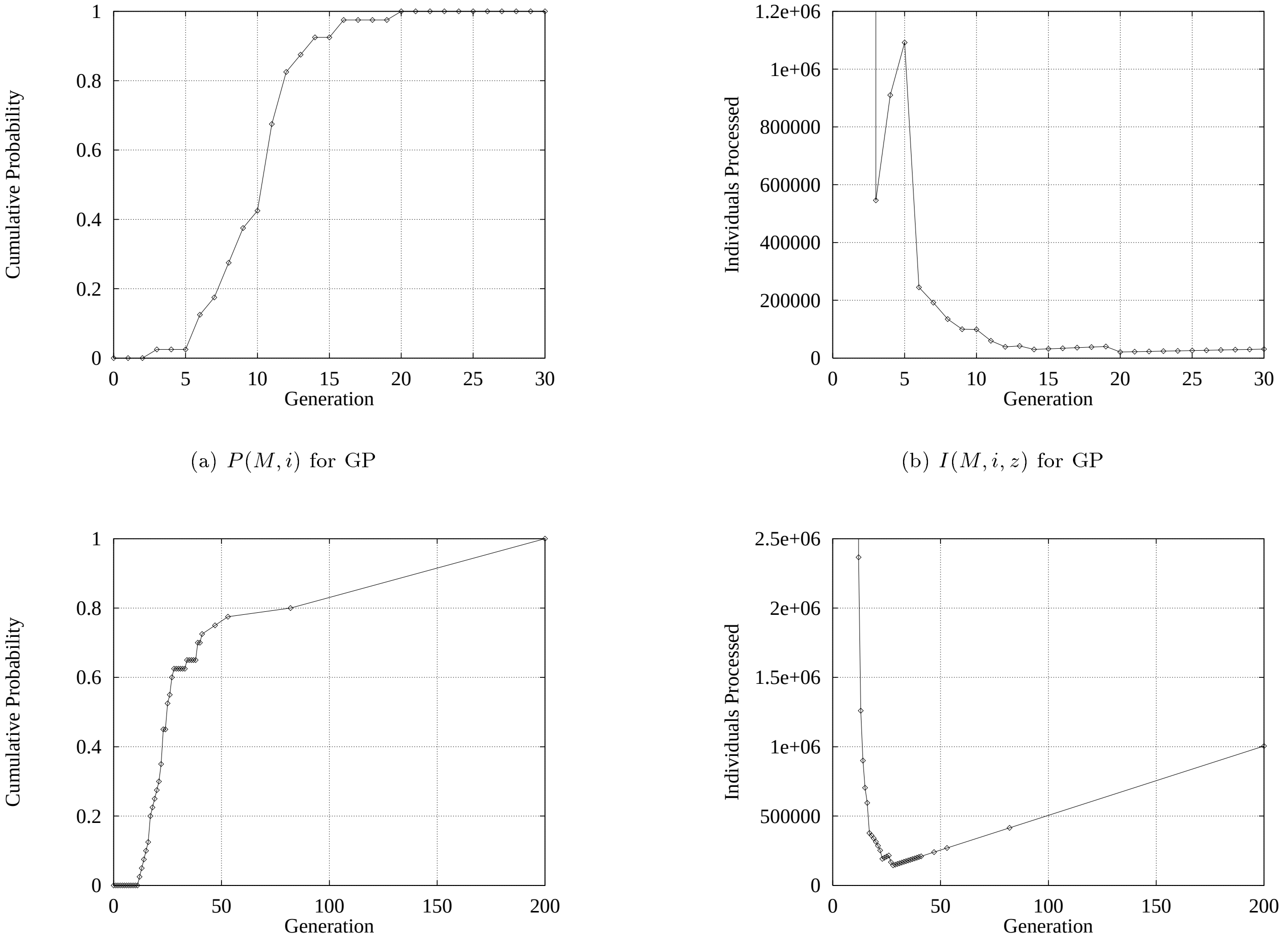
Fig. 4(a) shows $P(M,i)$ versus generations for GP. At Generation 20 the cumulative probability reaches unity. Compare this with Fig. 4(c) for GA—even at Generation 82 the cumulative probability has only reached 0.8. Figures 4(b) and 4(d) present the individuals processed against generation for GP and GA respectively. For GP, a single run carried to 20 generations yields $P(M,i) = 1$ and requires the processing of only 21,000 individuals. However, GA requires a minimum of 145,000 individuals to be processed—for Generation 28 with a $P(M,i)$ of 0.625.

6 Function Set Redundancy

Given the clear superiority of GP over GA in this example, our attention now focused on how this had been achieved, and whether further improvements in performance could be obtained by examining the agents that had been evolved.

We looked at the best individual in each of the 40 runs for GP and found that only four out of the nine functions available to the agents were utilised by the top individuals in all runs. Specifically we found that `forage` was used 79 times, `pigout` 124, `wander` 16 and `run` 25 times. The remaining five functions (*i.e.* `deposit`, `walk`, `nibble`, `turn` and `nop`) did not occur at all. This led to the logical reduction of the function set for GP. We restructured the GP-based agent to use a function set of {`forage`, `run`, `pigout`, `wander`} and a terminal set of {`0`, `1`, `2`} and obtained another 40 runs. Figures 5(a) and 5(b) present the performances $P(M,i)$ and $I(M,i,z)$ of the GP for the trimmed-down function and terminal set.

Following the same lines we inspected the strings in our GA results of those agents that obtained a fitness of 65. We found that only five functions were used: `pigout` 160 times, `forage` 130, `wander` 16, `run` 12 and `walk` twice. Accordingly we reduced the function set for our GA-based agent; this resulted in a 50-bit string (*cf.* a 60-bit string used in the initial experiment to incorporate all ten functions). The new



(a) $P(M,i)$ for GP

(b) $I(M,i,z)$ for GP

(c) $P(M,i)$ for GA

(d) $I(M,i,z)$ for GA

Figure 4: Initial results

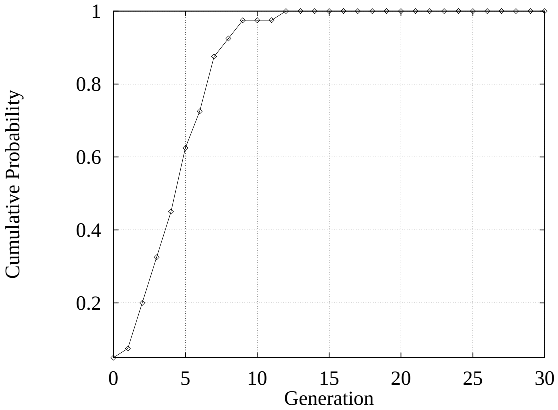
$P(M,i)$ and $I(M,i,z)$ for GA are presented in Figures 5(c) and 5(d), respectively.

Clearly, with a more focused set of functions available to the agents, we find that both GP and GA show improved performances; in particular, GA has now come almost level with GP. For $P(M,i)$, both GP and GA now reach unity, but in Gens 12 and 16 respectively. For $I(M,i,z)$, GP needs to process only 13,000 individuals (Generation 12) while GA requires only slightly more at 17,000 (Generation 16).

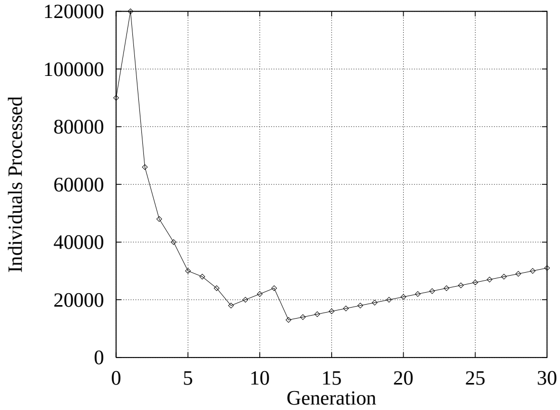
7 Conclusions & Further Work

Our findings clearly show that, at least for the simple autonomous agents and limited problem size examined, a well-focused set of functions can greatly enhance the performance of both genetic programming (GP) and genetic algorithm (GA) approaches to agent evolution. Under these conditions, both GP and GA appear equally effective in evolving agent-based software; but, where the problem is perhaps less well understood, and the functions used are not as tightly constrained, GP may well outperform GA.

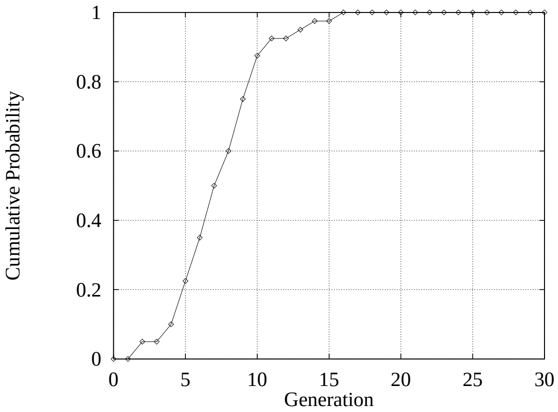
The work presented here could be extended by fully implementing Maskell & Wilby's Model-1 and Model-2 [4] (the latter provides a dynamically varying environment and extended agent functionality), and comparing GA and GP performance. In addition, larger problems of a similar nature could be examined to assess how well our conclusions scale with problem size. However, it is anticipated that most of the second author's future work will be devoted to evolving increasingly complex agents for telecommunications routing and restoration using GP.



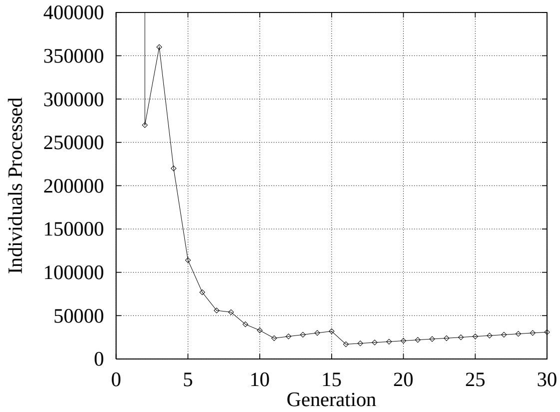
(a) $P(M, i)$ for GP



(b) $I(M, i, z)$ for GP



(c) $P(M, i)$ for GA



(d) $I(M, i, z)$ for GA

Figure 5: Reduced function set results

References

[1] Goldberg, D.E.: *Genetic Algorithms in Search, Optimization and Machine Learning* Addison-Wesley, 1989

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

[3] Kirkwood, I.M.A., Shami, S.H. & Sinclair, M.C.: Discovering simple fault-tolerant routing rules using genetic programming *Proc. Intl. Conf. on Artificial Neural Networks and Genetic Algorithms* Norwich, UK, 1997

[4] Maskell, B. & Wilby, M.: Evolving software agent behaviours *GLOBECOM'96* London, 1997, pp.90–94

[5] Steenstrup, M. (ed); *Routing in Communication Networks* Prentice Hall, 1995

[6] Ash, G.R.: Dynamic network evolution, with examples from AT&T's evolving dynamic network *IEEE Communications Magazine v33 n7* July 1995

[7] *IEE Colloquium on Intelligent Agents and their Applications, Digest No.: 96/101* London, April 1996

[8] Zongker, D. & Punch, B.: *lil-gp 1.0 User's Manual* Michigan State University, 1995

[9] Grefenstette, J.J.: *A User's Guide to GENESIS Version 5.0* 1990