Dormant Program Nodes and the Efficiency of Genetic Programming

David Jackson
Dept. of Computer Science
University of Liverpool
Liverpool L69 3BX, United Kingdom
Tel. +44 151 794 3678
d.jackson@csc.liv.ac.uk

ABSTRACT
In genetic programming, there is a tendency for individuals in a population to accumulate fragments of code – often called introns – which are redundant in the fitness evaluation of those individuals. Crossover at the sites of certain classes of intron cannot produce a different fitness in the offspring, but the cost of identifying such sites may be high. We have therefore focused our attention on one particular class of non-contributory node that can be easily identified without sophisticated analysis. Experimentation shows that, for certain problem types, the presence of such dormant nodes can be extensive. We have therefore devised a technique that can use this information to reduce the number of fitness evaluations performed, leading to substantial savings in execution time without affecting the results obtained.

Categories and Subject Descriptors
D.1.2 [**Programming Techniques**]: Automatic Programming; I.2.2 [**Artificial Intelligence]** Automatic Programming – *program synthesis*; I.2.6 [**Artificial Intelligence**] Learning – *induction*.

General Terms
Algorithms, Performance, Experimentation.

Keywords
Genetic programming, efficiency, intron, dormant node, fitness-preserving crossover.

1. INTRODUCTION
In genetic programming, many members of the evolving population will contain sequences of program code which make no contribution to the fitness of those individuals. These code fragments are commonly called *introns*, although there are researchers who dislike the term [6]. Introns have come under intense research scrutiny, in particular for the role they might play in the phenomenon of *code bloat* – the undesired increase in code size during evolution (e.g. [7,8,10-13]), but also for other reasons such as their impact on the effectiveness of genetic programming systems at finding solutions [2,9].

For certain types of intron, crossover that takes place at the site of the non-contributing code cannot alter the fitness that is propagated to the generated offspring. In a paper on the genetic programming of data structures, Langdon [5] pointed out that it might even be possible to avoid invoking the fitness function used to evaluate those offspring, possibly resulting in considerable savings in execution time. The remark was made almost as an aside, and Langdon presented no experiments or results to support the idea.

A difficulty with this suggestion is that the computational costs associated with identifying the introns in the first place can render such an approach impractical. For genetic programming problems involving, say, loops or recursion, the analysis required may be intractable [3,13]. For other problems, such as the evolution of a multiplexer, where individuals can be exhaustively tested with all combinations of inputs, it may be possible to identify introns by systematically replacing each sub-tree with a nullifying or negating operation to see if the fitness value alters [1,7]. Such an approach may be feasible but computationally highly expensive.

In this paper, we explain how the costs of identification can be minimised by restricting ourselves to a subset of the possible intron types. We describe a simple implementation method for identifying such program nodes, and then investigate how widespread this limited category of nodes really is in a population. Exploitation of these results leads us to a mechanism for reducing the number of fitness evaluations that are necessary, as per Langdon's suggestion, and we present results for the effectiveness of the technique.

2. DORMANT NODES
Although the term *intron* has been widely used in the genetic programming literature to denote pieces of program code that make no contribution to the fitness of an individual, the term is often imprecisely defined. Nordin *et al* [9] addressed this by defining five main categories of intron. Type 1 introns, for example, are code segments in which crossover never changes the behaviour of the program for any input in the problem domain, while Type 2 introns are code segments where crossover never changes the behaviour of the program for any of the fitness cases.

GECCO'05, June 25-29, 2005, Washington, DC, USA.

Soule and others [6,13] abandoned the use of the word intron in their own categorisation defining the viability and operability of program nodes and sub-trees. In their usage, an *inoperative* node is one in which replacement of the sub-tree rooted at that node by a null operation (no-op) will not change the program's output. Similarly, a node is said to be *inviable* if there does not exist a sub-tree which will change the program's output when substituted for that node.

For our own purposes, the key information we require is whether a node is executed or not for the set of test cases used to evaluate a program. We are less interested in making broader statements regarding the potential contribution of such nodes. Indeed, it is not always clear that a simple division of nodes into executed and non-executed camps sits comfortably with existing taxonomies. Nodes that are not executed during a fitness evaluation are certainly of the Type 2 variety as defined by Nordin *et al*, but many may also be of the Type 1 variety. In other words, the reason a node is not executed may be either because the test cases are not sufficient, or because the node is unreachable in all circumstances. Similarly, the categorisation of Soule *et al* seems to be defined in terms of the problem domain, rather than in terms of specific execution cases. Inviable code is often unexecuted code, but the two are not synonymous.

For these reasons, we introduce a new terminology as follows:

Definition: A *dormant* node is a program node that is not executed for any of the cases involved in evaluating the fitness of the individual. A non-dormant node is said to be *active*. It follows from this that if the root node of a sub-tree is dormant, then the whole sub-tree is dormant, i.e. none of the nodes in that sub-tree is executed.

From this definition, it will be realised that the decision as to whether a node is denoted dormant or active can be based purely on information gathered dynamically, and requires no sophisticated analysis of the containing program. To record this information, we adopt a *marking* method similar to that used by Blickle and Thiele [1]. In their work, Blickle and Thiele used marking as a means for approximating the amount of 'redundancy' present in a program. They were interested in the effects of *avoiding* redundant crossovers, and so presented no timings for the costs and savings involved when these crossovers are allowed to proceed.

Since we need to record possible dormancy for all nodes of all trees present in the genetic programming population, every individual is allocated what we call a *visit tree*. This is of the same length and structure of the individual's code tree. When an individual is presented to the fitness function for evaluation, all of the nodes in its visit tree are initialised to a pre-defined NOT-VISITED value. Whenever the fitness function causes a node of the program tree to be evaluated, the corresponding node of the visit tree is set to VISITED. To implement this we use a visit tree pointer that mirrors the navigation of the code tree pointer as the individual's program tree is traversed. Care must be taken to ensure that the visit tree pointer is properly updated when sub-tree arguments are skipped during, for example, the evaluation of an IF-THEN-ELSE node.

Since each visit tree node records only binary information (visited or not), then it would be possible to encode it as a single bit. This was, in fact, the suggestion made by Blickle and Thiele. However, although it is certainly more memory efficient, the bit manipulation and testing operations it necessitates make it far less time efficient, and so we settled on the use of a complete byte per tree node.

3. DORMANCY PREVALENCE

Once the visit tree is in place and functioning, it is a simple matter to build up information about the amount of dormant and active nodes present in a population as it evolves. To illustrate this, we will make use of a maze navigation problem. One of the reasons for choosing this particular problem is that it has already been used as the subject for intron research by Soule *et al* [6,13].

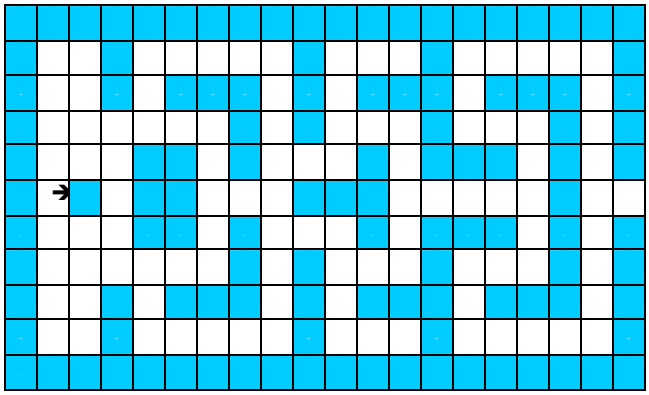

Fig. 1. Pre-defined maze used in the maze navigation problem

In our slightly adapted version of the maze problem, the objective is to navigate successfully not one but a number of mazes (we used twenty). One of these mazes has a pre-defined topology; the others are generated at random. The pre-defined maze is shown in Figure 1; this is identical to the maze used by Soule *et al*, except for the addition of a single exit square in the right-hand wall. The initial position and orientation of the entity to be guided through the maze is indicated by the arrow.

The other parameters for the problem are presented in Table 1.

The agent can turn left or right, move forward or backward, and test whether there is a wall ahead or not. A no-op terminal does nothing except to expend an instruction cycle. Decision making is via an if-then-else function, whilst iteration is achieved via a while function, which repeatedly evaluates the second sub-tree argument whilst the first argument evaluates to true. The PROGN2 function is simply a connective which evaluates each of its two arguments in turn. For a given maze, program fitness is measured in terms of how close the agent gets to the exit: zero fitness indicates escape from the maze. Navigation continues until a maze is successfully completed, or an upper bound of 3000 instruction cycles is reached.

Given the nature of the function set, there is the potential for dormancy to be present in the population. Even over the completion of all twenty mazes, the possibility remains that there are branches of if-then-else clauses that do not get executed, or

while loops for which the predicates always evaluate to false. The first thing to determine is how far-reaching this dormancy is.

Table 1. Tableau for the maze navigation problem

Objective	To navigate a set of mazes
Terminal set	forward, back, left, right, no-op, wall-ahead, no-wall-ahead
Function set	if-then-else, while, progn2
Initial population	Ramped half-and-half, no duplicates
Evolutionary process	Steady-state; 5-candidate tournament selection
Fitness cases	20 mazes: 1 pre-defined, 19 random
Fitness	Closest distance to exit (0-18), summed over all mazes
Restrictions	Programs timed-out after 3000 instructions
Success predicate	Zero fitness (all mazes navigated)
Other parameters	Pop size=500; Gens=51; prob. crossover=0.9; no mutation; prob. internal node used as crossover point=0.9

Figure 2 depicts a graph showing the total number of nodes present in the population as it evolves during a typical run, together with the total number of active (non-dormant) nodes. The expected phenomenon of code bloat is evident, with the population size rising from 12,000 to 1.5 million nodes in just 17 generations, and a more gentle ascent to 2.4 million nodes thereafter. By comparison, the number of active nodes seems to reach a ceiling after generation 17, hovering at just above the 500,000 node level. The amount of dormancy present in the population is therefore extensive: by generation fifty, 76% of all nodes are dormant; that is, the population contains roughly 1.8 million nodes that are never executed and so make no contribution to fitness.

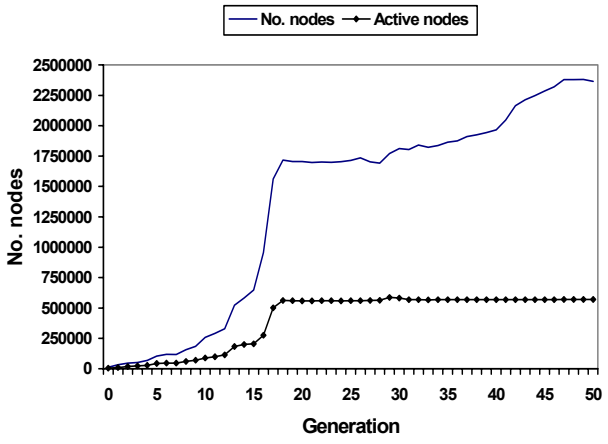

Fig. 2. Total and active nodes in population for one run of maze problem

The given run profile is not an extreme example. Figure 3 shows the percentage of dormancy present in the population at the end of each of a sequence of 20 runs. Solutions were obtained in two of the runs: run 3 and run 18. In run 3, a solution was not found until generation 49, but in run 18 a solution was found somewhat earlier at generation 34. The termination of the run at that point may account for the lower level of dormancy built up in the reduced time. In general, however, the amount of dormancy is extensive, in some cases accounting for more than 80% of all nodes.

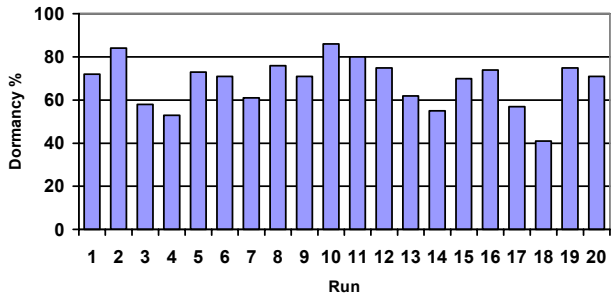

Fig. 3. Percentage of dormancy in 20 runs of maze problem

The marking method we have described makes it easy to determine when crossover cannot possibly lead to a change in fitness for the newly-created offspring. During crossover, a sub-tree of parent P1 is replaced by a selected sub-tree of parent P2 to create a new child. If P1's sub-tree is known to be dormant, then the newly-inserted sub-tree must also be dormant, and so fitness cannot alter. This is true even if the transplanted sub-tree was active in the original parent P2, and perhaps even of immense operative value there. Note that the converse can also happen in crossover, with 'sleeping' nodes being awakened upon transfer to another individual. Indeed, one of the reasons the term 'dormant' was chosen was to reflect this context-dependency.

We use the term fitness-preserving crossover (FPC) to denote the situation in which crossover is made at a dormant node and which therefore cannot affect fitness. This is to distinguish it from the more commonly used term neutral crossover, which refers to the creation of a child which, upon evaluation, is found to have the same fitness as its parent. By contrast, FPCs are determined prior to (and, as we shall see, obviate) any fitness evaluation. Moreover, there may be crossovers which do not take place at dormant nodes and yet still lead to equivalent fitness; in other words, FPCs are a subset of neutral crossovers. It should also be noted that saying FPCs do not alter fitness is not the same as saying that they are worthless: they may play a valuable role in propagating useful genetic material.

Since dormancy has been shown to be so prevalent, it might also be expected that FPCs occur frequently during evolution. This is borne out in Figure 4, which shows the growth in the percentage of FPCs taking place during the typical run we referred to earlier. In generation fifty, 74% of all crossovers are FPCs. Over the whole run, 64% of all crossovers are FPCs. As before, we can widen the picture to our sequence of 20 runs (Figure 5). Again, solutions were obtained in runs 3 and 18. In all but two of the other runs, the FPC count is above 50%, and in one run reaches higher than 80%.

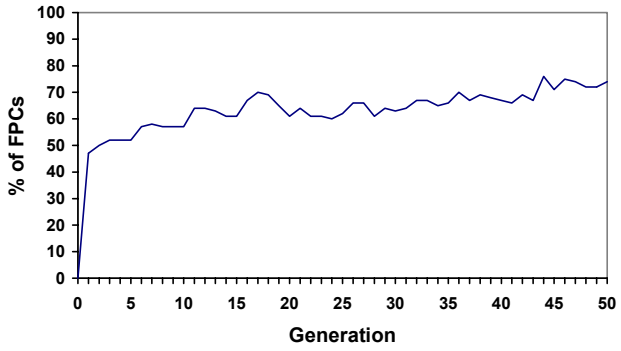

Fig. 4. Percentage of fitness-preserving crossovers in single run of the maze problem

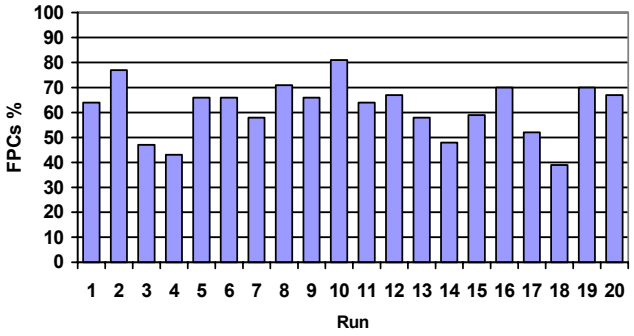

Fig. 5. Percentage of fitness-preserving crossovers in 20 runs of maze problem

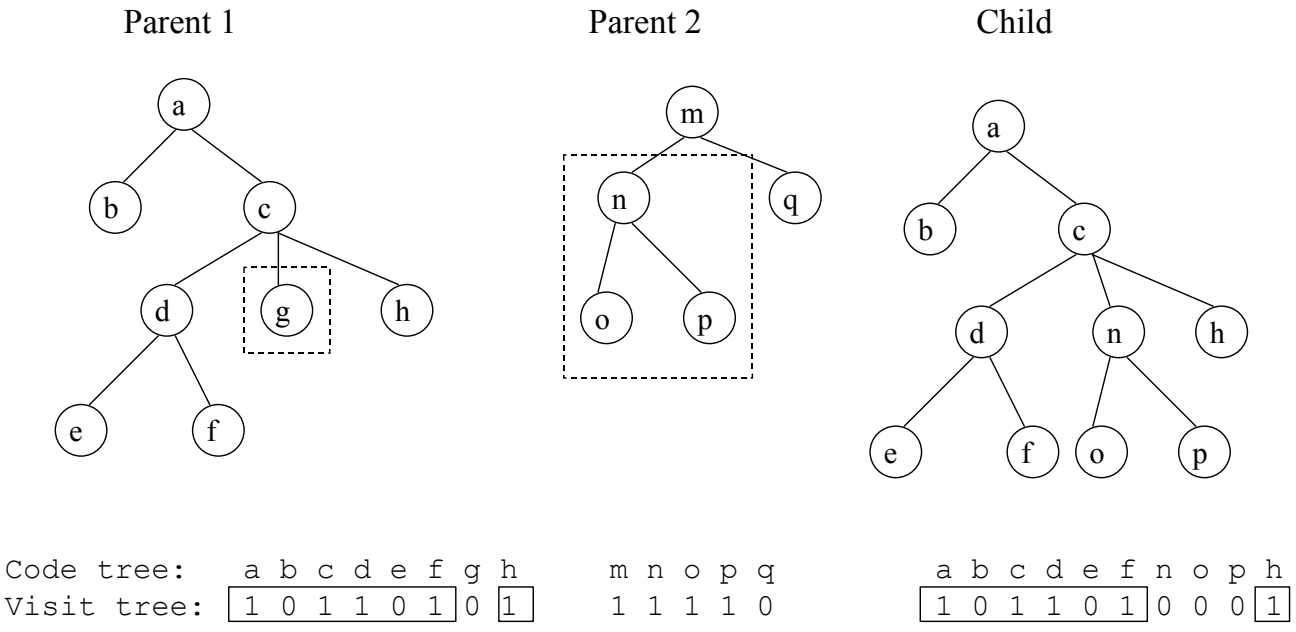

Fig. 6. Manipulation of visit trees during crossover

4. EVALUATION AVOIDANCE

Since the fact that a crossover is fitness-preserving is known in advance of fitness evaluation, an obvious question is whether the evaluation need be performed at all. Clearly, the child can simply be assigned the same fitness as its parent. However, this leaves us with a problem: without evaluation, how can a visit tree be created for the new individual which is useful for future determination of FPCs?

The solution lies in the recognition that, like the individual's code tree, its visit tree can also be created via an analogous crossover operation. Figure 6 shows how this works.

In this particular crossover, node *g* of Parent 1's code tree is replaced by the sub-tree *n-o-p* of Parent 2 to create the child on the right. However, the flattened form of the visit tree for Parent 1 shows that node *g* is a dormant node, and this is therefore an FPC. The new visit tree for the child can be derived from its parent visit

trees simply by replacing the appropriate nodes with a new sequence that is of the same length of the sub-tree brought in from Parent 2. Moreover, since the new sub-tree must be dormant, all of the nodes in the inserted sequence can and should be initialised to dormant (zero in the diagram), rather than copied from Parent 2.

This operation is obviously an overhead, and is in addition to the overheads of visit tree creation, initialisation and maintenance already described. The next question is whether these overheads are outweighed by the savings that can be achieved through non-evaluation of individuals created via FPC. Table 2 shows the elapsed times for 20 runs of our maze navigation problem, using both conventional fitness evaluation and evaluation for non-FPC individuals only. In both cases, the genetic programming system was initialised with the same random number seed, to ensure that the evolutionary process and the results obtained in terms of best programs etc. were identical. The timings were performed on a PC with a 2.8GHz Pentium 4 processor and 512MB dual DDR RAM.

Table 2. Execution times for conventional and non-FPC evaluation approaches in maze problem

Fitness evaluation performed	**Elapsed time for 20 runs (secs)**
Conventional	250
Non-FPC only	113

It can be seen from the table that the evaluation avoidance technique we have described leads to a 55% saving in elapsed execution time.

5. OTHER PROBLEMS

The technique described in this paper has been applied to a number of problem domains, with similar significant results. To take another example, consider the well-known Santa Fe trail problem, in which an artificial ant must be guided along a trail of food particles [4]. Table 3 gives the parameters for this problem.

Although this problem has an IF-THEN-ELSE statement, it does not contain any explicit loops or other constructs in which code can remain unexecuted. It might be presumed, therefore, that the scope for dormancy is limited. Experimental investigation reveals, however, that dormancy is in fact more extensive than it is in the maze problem, as can be seen in the single run depicted in Figure 7, and in the sequence of runs shown in Figure 8.

In the majority of runs, dormancy exceeds 80%, and sometimes reaches the mid-nineties. As with the maze problem, the number of active nodes in the population as a whole tends to reach a plateau while the population size continues to climb. Figure 9 shows how the number of fitness preserving crossovers for one run of this problem climbs steadily until it reaches 93% in generation fifty, while Figure 10 shows the total number of FPCs in each of a sequence of 20 runs of this problem.

Table 3. Tableau for the artificial ant problem

Objective	To evolve a program that guides an ant along a trail of food particles
Terminal set	left, right, move
Function set	if-food-ahead, progn2, progn3
Initial population	Ramped half-and-half, no duplicates
Evolutionary process	Steady-state; 5-candidate tournament selection
Fitness cases	One: the Santa Fe trail
Fitness	Number of food pellets (0-89) not found by the ant
Restrictions	Programs timed-out after 600 steps (left, right or move)
Success predicate	Zero fitness (all food found)
Other parameters	Pop size=500; Gens=51; prob. crossover=0.9; no mutation; prob. internal node used as crossover point=0.9

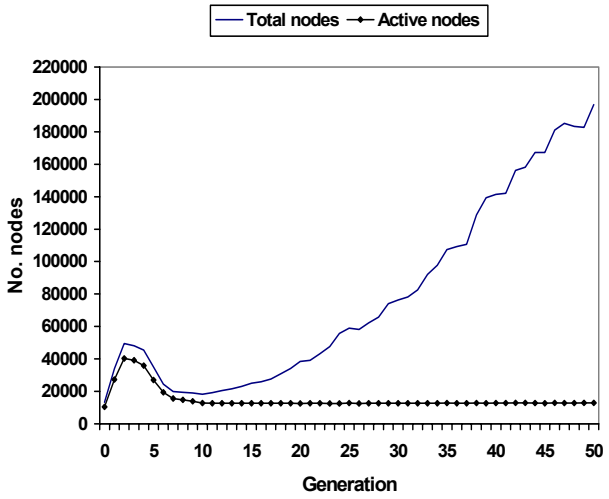

Fig. 7. Level of dormancy in one run of the ant problem

Table 4 gives the timings for the Santa Fe problem, taken over 200 runs. Coincidentally, the time for 200 runs of this problem is the same as that taken for 20 runs of the maze navigation problem, but the time for the non-FPC approach is lower. This latter figure represents a 62% saving in execution time.

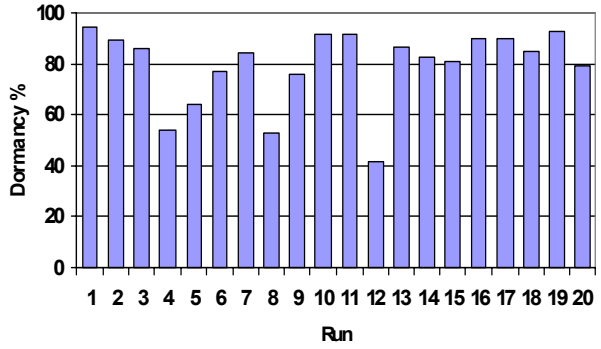

Fig. 8. Total dormancy levels in 20 runs of the ant problem

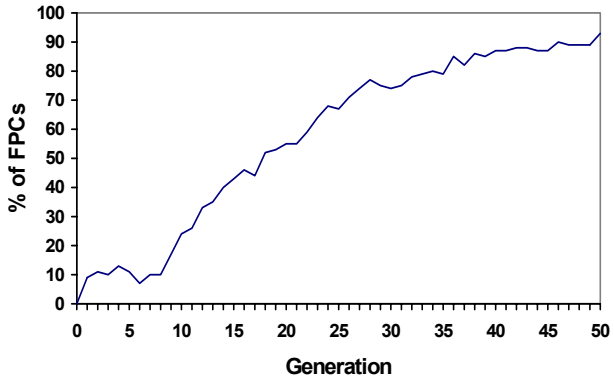

Fig. 9. Percentage of fitness preserving crossovers in one run of the Santa Fe artificial ant problem

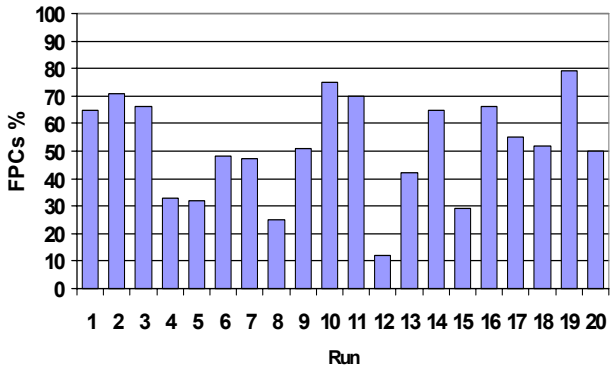

Fig. 10. Percentage of FPCs in 20 successive runs of the ant problem.

Table 4. Execution times for conventional and non-FPC evaluation approaches in Santa Fe problem

Fitness evaluation performed	Elapsed time for 200 runs (secs)
Conventional	250
Non-FPC only	95

6. CONCLUSIONS

Although dormant nodes do not equate to the full range of possible intron types that be present in a population, we have shown that they are certainly highly prevalent in certain classes of genetic programming problem. These problems are ones which make use of constructs, such as if statements and while loops, that permit the possibility of non-traversed paths in the program tree. It would seem that the presence of only one such construct in the function set is sufficient to entail significant dormancy.

Where dormancy is so prevalent, then so too are fitness-preserving crossovers. The mechanisms we have described for detecting when these occur, and the subsequent visit tree crossovers required to avoid fitness evaluation, are easily implemented. As we have demonstrated, the efficiency payoff can be substantial. Since program tree crossovers are not prevented, the results obtained remain unaffected. In the experiments described here, the only difference to the observer is an execution time that is less than half of that obtained using conventional techniques.

Future work will consist of further investigations into the role played by dormant nodes, with regard to performance, efficiency and code size in a variety of problem domains.

7. REFERENCES

[1] Blickle, T. and Thiele, L. Genetic Programming and Redundancy. In *Genetic Algorithms within the Framework of Evolutionary Computation* (Workshop at KI-94), Hopf, J. (ed), Saarbrücken, 1994, 33-38.

[2] Carbajal, S. and Martinez, F. Evolutive Introns: A Non-Costly Method of Using Introns in GP. *Genetic Programming and Evolvable Machines 2*, 2 (June 2001) 111-122.

[3] Iba, H. and Terao, M. Controlling Effective Introns for Multi-Agent Learning by Genetic Programming. In *Proc. Genetic and Evolutionary Computation Conf. (GECCO)*, 2000, 419-426.

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

[5] Langdon, W.B. Data Structures and Genetic Programming. In *Advances in Genetic Programming*, vol. 2, Angeline, P.J. and Kinnear, K.E. (eds), MIT Press, Cambridge, MA, 1996, 395-414.

[6] Langdon, W.B., Soule, T., Poli, R. and Foster, J.A. The Evolution of Size and Shape. In *Advances in genetic programming*, vol. 3, Spector, L. et al (eds), MIT Press, Cambridge, MA, 1999, 163-190.

[7] Luke, S. Code Growth is Not Caused by Introns. In *Late Breaking Papers, Proc. Genetic and Evolutionary Computation Conf. (GECCO)*, 2000, 228-235.

[8] Miller, J. What Bloat? Cartesian Genetic Programming on Boolean Problems. In *Late Breaking Papers, Proc. Genetic and Evolutionary Computation Conf. (GECCO)*, 2001, 295-302.

[9] Nordin, P., Francone, F., and Banzhaf, W. Explicitly Defined Introns and Destructive Crossover in Genetic Programming. In *Advances in Genetic Programming*, vol. 2, Angeline, P.J. and Kinnear, K.E. (eds), MIT Press, Cambridge, MA, 1996, 111-134.

[10] Smith, P. and Harries, K. Code Growth, Explicitly Defined Introns and Alternative Selection Schemes. *Evolutionary Computation 6*, 4 (1998) 339-360.

[11] Soule, T. Exons and Code Growth in Genetic Programming. In *EuroGP 2002, Lecture Notes in Computer Science vol. 2278*, Foster, J.A. et al (eds), Springer-Verlag, Berlin Heidelberg, 2002, 142-151.

[12] Soule, T., Foster, J.A., and Dickinson, J. Code Growth in Genetic Programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, Koza, J.R. et al (eds), MIT Press, Cambridge, MA, 1996, 215-223.

[13] Soule, T. *Code Growth in Genetic Programming*. PhD Thesis, University of Idaho, 1998.