Genet Program Evolvable Mach (2010) 11:89–121
DOI 10.1007/s10710-009-9086-1

ORIGINAL PAPER

The identification and exploitation of dormancy in genetic programming

David Jackson

Received: 28 January 2009 / Revised: 10 June 2009 / Published online: 27 June 2009

Abstract In genetic programming, introns—fragments of code which do not contribute to the fitness of individuals—are usually viewed negatively, and much research has been undertaken into ways of minimising their occurrence or effects. However, identification and removal of introns is often computationally expensive and sometimes intractable. We have therefore focused our attention on one particular class of intron, which we refer to as dormant nodes. Mechanisms for locating such nodes are cheap to implement, and reveal that the presence of dormancy can be extensive. Once identified, dormancy can be exploited in at least three ways: improving execution efficiency, improving solution-finding performance, and simplifying program code. Experimentation shows that the gains to be had in all three cases can be significant.

Keywords Genetic programming · Introns · Efficiency · Performance · Simplification

1 Introduction

Many of the individuals in a Genetic Programming (GP) population will contain program fragments which, for a variety of reasons, make no contribution to the fitness of those individuals. The label conventionally given to such a fragment is an *intron*. Much research has been concentrated on the behaviour and effects of introns, primarily because they tend to accumulate within individuals rapidly, leading to the phenomenon of code bloat. In general, code bloat is viewed as an undesirable side-effect of the evolutionary process because of the large, unwieldy and unfathomable programs that evolve, clogging up memory and slowing execution. A large part of the aforementioned research has therefore focused on understanding the role that introns may or may not play in code

D. Jackson (✉)
Department of Computer Science, University of Liverpool, Liverpool L69 3BX, UK
e-mail: djackson@liverpool.ac.uk

bloat so that it may be more effectively controlled [1–10]. However, there is also research which stresses the more positive aspects of intron presence, to the extent that it may sometimes be beneficial deliberately to inject them into the gene pool [11, 12].

In this paper we wish to consider the ways in which recognition of introns can be used to our advantage. Specifically, we will focus on a subset of introns which we refer to as *dormant nodes*. In the general case, introns can be difficult to locate within a program without extensive and costly analysis; it may even be impossible to determine the presence of some types of intron, particularly when loops and recursion are involved [3, 13]. One of the advantages of dealing with dormant nodes is that they are easily identified; moreover, they correspond to the vast majority of introns usually found within evolved programs. The mechanisms for locating such nodes will be described later.

Once identified, dormancy can be exploited in at least three ways. The first of these involves improving the execution efficiency of a GP system. As we shall see, a map of where dormant nodes lie within a program can, in many cases, act as an indicator as to when the usual method for assessing the fitness of programs can be circumvented. Secondly, it may be possible to use our knowledge of the dormancy within programs to increase the success rate of GP systems in finding solutions. We will also try to ascertain what it is about certain problems that makes such performance improvements possible. Finally, a dormancy map offers us a means of simplifying evolved programs, which are often huge and complex.

Before considering each of these techniques in turn, we need to define precisely what is meant by dormant nodes, and also how we go about locating them.

2 Introns and dormant nodes

Although the term *intron* has been widely used in the GP 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. [11] addressed this by defining five main categories of intron, as follows:

Type 1: *Code segments in which crossover never changes the behaviour of the program for any input in the problem domain*. In other words, replacing that segment with a different program sub-tree cannot possibly affect the behaviour of the individual as a whole. For example, in (MULT 0 (ADD X X)) the (ADD X X) sub-tree is a Type 1 intron: any change to it will still result in a zero value for the whole tree. Other Type 1 introns may be far more difficult to detect. Consider code such as the following:

```
if (x < 5)

then {S1}

else {  if (x < 0)

        then {S2}

     }
```

Here, S2 is clearly unreachable, and so whatever code we might replace it with will not affect the program. However, recognizing that the outer predicate ($x < 5$) impacts upon the inner predicate ($x < 0$) is a difficult analytical problem in the general case.

Type 2: *Code segments where crossover never changes the behaviour of the program for any of the fitness cases*. For example, in (IF (AND X1 X2) (OR D0 D1) (NOT D1)), the (OR D0 D1) sub-tree will be executed only when X1 and X2 are both true. If the input test data is such that this is never the case for X1 and X2, then (OR D0 D1) is a Type 2 intron. Note that this categorisation is very specific to a given test data set: it says nothing about the problem domain as a whole unless input coverage is exhaustive.

Type 3: *Code segments which cannot contribute to the fitness and where each node can be replaced by a no-operation without affecting the program for any input in the problem domain*. As an example, consider (PROGN2 FORWARD (PROGN2 LEFT RIGHT)). For some problems, the (PROGN2 LEFT RIGHT) sub-tree has no overall effect, and can therefore be replaced by a null operation. This differs from Type 1 and Type 2 introns in that replacing the sub-tree with alternative code could still alter program behaviour. Note too that this example does not apply to those problems in which functions have side-effects, e.g. the Santa Fe ant problem, where each move consumes a unit of simulated time.

Type 4: *Code segments which cannot contribute to the fitness and where each node can be replaced by a no-operation without affecting the program for any of the fitness cases*. As with Type 2 introns, these are similar to the immediately preceding category, but are determined upon program execution with the input data set only.

Type 5: *More continuously defined intron behaviour where nodes are given a numerical value of their sensitivity to crossover*. These are common in symbolic regression type problems, where values can become so large or small in magnitude that they make no significant contribution. An example is (DIV (SUB X 2)(EXP (EXP 20))), where the (EXP (EXP 20)) value is so huge that arithmetic rounding almost always causes zero to be returned as the result of the division, no matter what the numerator expression is. In other words, (SUB X 2) is a Type 5 intron here.

The above is probably the most detailed taxonomy of introns that has been proposed, although a number of other categorisations and terminologies have been devised. Soule and others [3, 14], for example, rejected the use of the word 'intron' because of its inaccuracy as an analogue for the biological term. In their alternative nomenclature, 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 the purposes of this paper, the key determination we need to make is whether or not a given program node is executed for the set of test cases used to evaluate a program. What makes this difficult to slot comfortably into existing taxonomies is that we are less interested in making broader statements about the *potential* contribution of such nodes. For example, 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. Conversely, there are Type 2 introns which are classed as such not merely by virtue of remaining unexecuted. For example, in (MULT (ADD X Y) (SUB X Y)), the (ADD X Y) sub-tree is a Type 2 intron if it turns out that (SUB X Y) is zero in all test cases. In the general case, however, it is likely that the (ADD X Y) sub-tree will be evaluated at some point, since it is not possible to reason *a priori* that it need not be.

Other taxonomies are associated with similar difficulties. For example, the categorisation of Soule et al. appears 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 definition 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. This is because a root node represents a function, and the sub-trees represent the function arguments. If the function is never executed, then its arguments are never evaluated. It is interesting to note that similar reasoning has been applied to other forms of intron, e.g. Soule and Foster show formally that the descendants of an inviable node are themselves inviable [15].

It may also be appreciated that the decision as to whether a node is deemed dormant or active can be based purely on information gathered dynamically, and requires no sophisticated analysis of the containing program. To derive this information, we adopt a *marking* method similar to that used by Blickle and Thiele [16], who used it as a means for approximating the amount of ‘redundancy’ present in a program.

Since we need to record possible dormancy for all nodes of all trees present in the GP population, every individual is allocated what we call a *visit tree*. This has the same size and structure as 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.

3 The extent of dormancy

Now that we have more precisely specified the sub-class of introns in which we are interested, we are in a position to discuss the extent to which they appear within program code. The visit trees mentioned above can be used to build up this information for each individual and for the population as a whole as evolution proceeds. We will consider this firstly in relation to the well-known Santa Fe trail problem [17], in which an artificial ant has to navigate a trail of food pellets within a fixed time period. The parameters for this problem as we have used them are presented in Table 1.

Since the function set for this problem contains an IF-THEN-ELSE construct (if_food_ahead) there is at least the potential for some sections of program code never to be executed for a given individual (i.e. to be dormant code). In Fig. 1 we chart the total number of program nodes present in the population during one run of the ant problem, together with the number of active (executed) nodes. As might be expected, the phenomenon of code bloat is clearly evident, with the total number of nodes in the population rising rapidly: from generation 10 onwards this number rises from 19,000 to 196,000 nodes—more than a ten-fold increase. In contrast, the number of active program nodes remains remarkably constant at about the 12,600 mark.

Table 1 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)
Other parameters	Population size = 500; Generations (total offspring/population size) = 51; prob. crossover = 0.9; no mutation; prob. internal node used as crossover point = 0.9

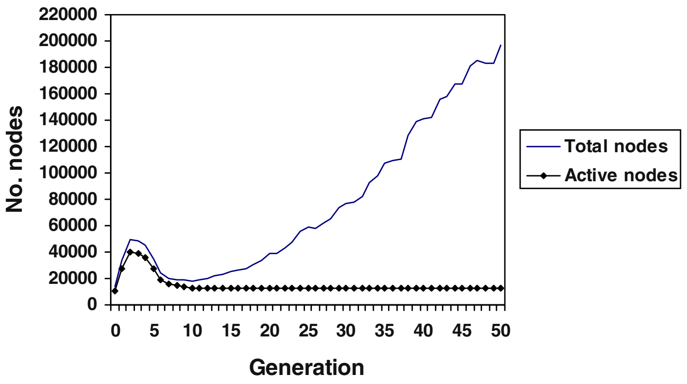


Fig. 1 Extent of dormancy in one run of the ant problem

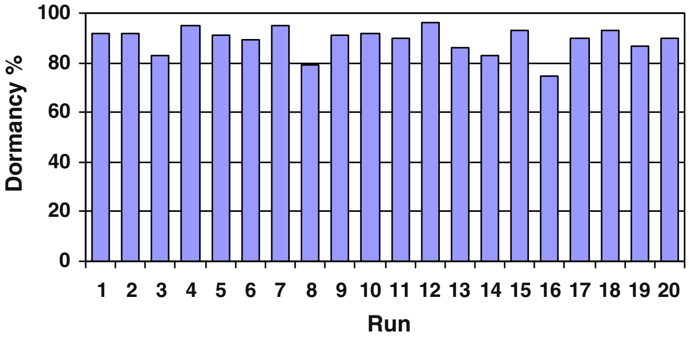


Fig. 2 Final dormancy levels in 20 successive runs of the ant problem

To put these figures another way, we can state that the level of dormancy in the population increases dramatically. At generation zero, approximately 20% of all program nodes are dormant, but by generation 50, the dormancy figure reaches 93%.

The given run profile is not an extreme example. Figure 2 shows the percentage of dormancy present in the population at the end of each of a sequence of 20 runs (in all cases, runs are not terminated prematurely, but are allowed to proceed until generation 50). It can be seen that in all of these runs the dormancy level never drops below 75%; indeed, in 13 of the runs it exceeds 90%, reaching a peak of 96% in run 12.

4 Efficiency

4.1 Approach

In a paper on the GP of data structures, Langdon [18] pointed out that, for certain forms of crossover taking place at the sites of known introns, it might be possible to avoid invoking the fitness function used to evaluate the new 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. As has already been mentioned, one of the difficulties of realising such techniques is in the prior identification of the introns, the cost of which can considerably outweigh the envisaged execution savings. By restricting ourselves to dormant nodes, where identification effort is low, a cost-effective approach may become more viable.

Using the marking technique we have described, it becomes easy to determine when crossover must lead to children whose fitness values are known prior to execution. During crossover, a sub-tree of one parent is replaced by a selected sub-tree of another parent to create a new child. In this and future discussion we will sometimes refer to the receiving parent as the *mother*, and the sub-tree donor parent as the *father*. If the mother'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 father, 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 must create a child which has a fitness value that is identical to that of its mother. 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 the need for) 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 that 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.

By definition, then, the occurrence of any FPC implies that the fitness evaluator need not be invoked to assess the fitness of the new offspring. This represents a saving in execution time, but for the proposed technique to be effective these occurrences must be frequent enough to ensure that they are not swamped by the computational costs of the marking technique. In the particular case of the ant problem referred to earlier, it might be expected that since dormancy has been shown to be so prevalent, then FPCs should occur frequently during evolution. This is borne out in Fig. 3, which shows the growth in the percentage of FPCs taking place during the typical run of that problem. In generation fifty, 93% of all crossovers are FPCs. Over the whole run, 59% of all crossovers are FPCs. As before, we can widen the picture to our sequence of 20 runs (Fig. 4). In all but three of the runs, the FPC count across the whole of the run is above 50%, and in one run reaches almost 80%.

For each of these FPCs, then, we can avoid invoking the fitness evaluation function, and simply assign the child the same fitness as its mother. A complication with this is that, for the approach to work as we have described, every individual must have a visit tree telling us which nodes are executed and which are not. If an

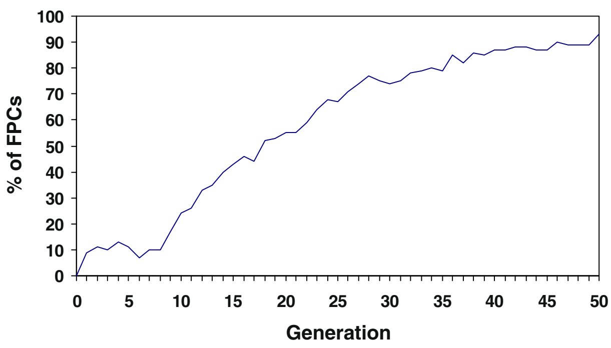


Fig. 3 Percentage of Fitness-Preserving Crossovers (FPCs) in one run of the artificial ant problem

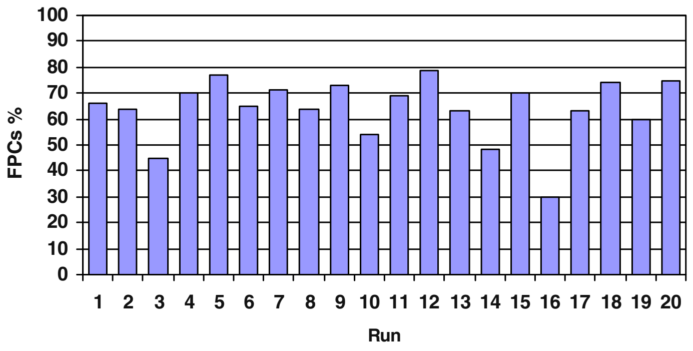
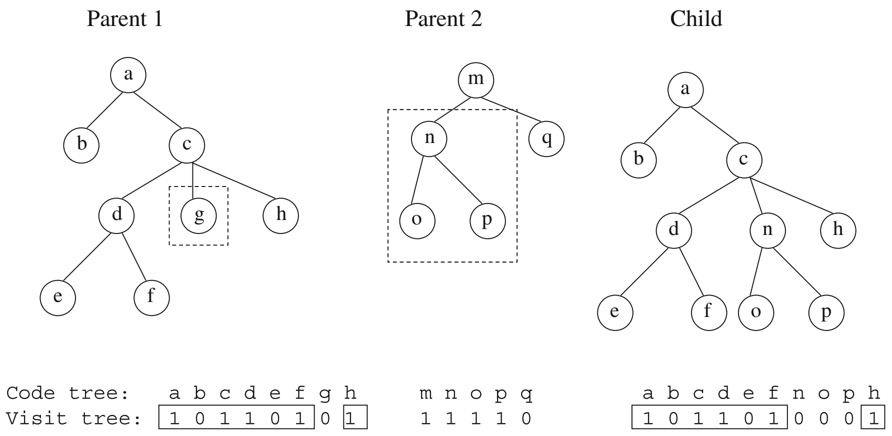


Fig. 4 Percentage of FPCs throughout each of 20 successive runs of the ant problem



```
Code tree:    a b c d e f g h      m n o p q         a b c d e f n o p h
Visit tree:  [1 0 1 1 0 1]0 [1]    1 1 1 1 0        [1 0 1 1 0 1]0 0 0[1]
```

Fig. 5 Manipulation of visit trees during crossover. Node *g* in *Parent 1* is being replaced by sub-tree *n-o-p* of *Parent 2* to produce the *child* on the right. The binary *visit tree* indicates whether a node is active (1) or dormant (0)

FPC-generated child is not executed, how do we create its visit tree for use in the recognition of future FPCs?

The answer lies in the recognition that, like the individual's code tree, its visit tree can also be created via an analogous crossover operation. Figure 5 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.

To compare the efficiency of a standard GP approach, in which fitness evaluation is performed for every crossover, with an approach in which fitness evaluation is

Table 2 Program node evaluations for the ant problem

Avg. nodes per run using conventional GP ($\times 10^6$)	29.5
Avg. nodes per run using non-FPC approach ($\times 10^6$)	11
Avg. reduction	62%
Max. reduction	83%
Min. reduction	21%
Statistically significant?	Yes

Table 3 Execution efficiency comparison for the ant problem

Fitness evaluation performed	Nodes evaluated in 100 runs ($\times 10^9$)	Elapsed time for 100 runs (s)
Conventional	2.95	136
Non-FPC only	1.1	48

performed only for those crossovers which are not FPCs, we can count the total number of program nodes that are executed in a sequence of runs. Table 2 gives these figures for 100 runs of the ant problem. For both approaches, the GP 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 non-FPC approach offers an average saving of approximately 62% of the number of nodes that need to be evaluated per run. In one of the runs the saving is as high as 83%. A statistical t-test indicates that the improvements are significant at the 99.9% confidence level.

How do these savings impact upon execution timings? Summing over the 100 runs, standard GP involves the execution of nearly 3 billion nodes, while the non-FPC-only approach requires just over a billion node firings. However, the overheads of visit tree creation, initialisation and maintenance already described may mean that these figures may not necessarily translate into equivalent run-time gains. Table 3 therefore also shows the elapsed execution times for 100 runs of both approaches. The timings were performed on a PC with a 2.8 GHz Pentium 4 processor and 512 MB dual DDR RAM. It will be seen that the non-FPC-only approach does in fact run in about one-third of the time of the standard approach.

4.2 Other problems

Providing convincing evidence of the efficacy of our approach requires that it be applied in more than just a single domain. Experiments have therefore been performed on the following problems:

4.2.1 Maze navigation

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. [3, 14]. In our slightly adapted version, the objective is to navigate successfully not one but a number of mazes (we

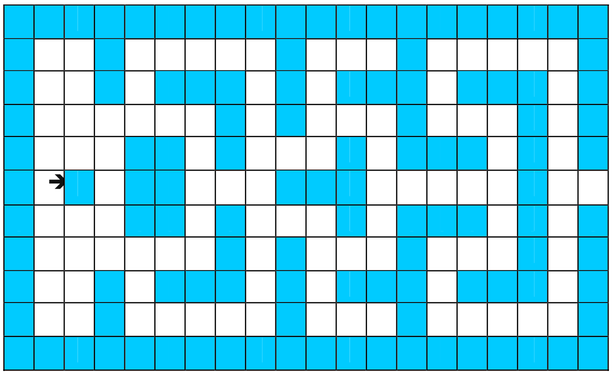


Fig. 6 Pre-defined maze used in the maze navigation problem

Table 4 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
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 3,000 instructions

used 20). One of these mazes has a pre-defined topology; the others are generated at random. The pre-defined maze is shown in Fig. 6; 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 4; for this and the other problems described here, only those parameters which differ from those of the ant problem in Table 1 are given.

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. 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 3,000 instruction cycles is reached.

4.2.2 Defence strategy

In this problem, the idea is to determine whether GP can be used to evolve winning battle strategies for a single defender facing multiple simultaneous attackers. The problem is couched in the form of the well-known ‘Space Invaders’ arcade game, and full details of our original approach to evolving such strategies can be found

Table 5 Tableau for the defence strategy problem

Objective	To win a sequence of wargames
Terminal set	Left, right, fire, attacked, target-left, target-right
Function set	If-then-else, progn2, progn3
Fitness cases	50 Randomised games
Fitness	Based on proximity of missiles to attackers, with penalties for defender being destroyed and attackers landing

elsewhere [19]. For the research described here, the relevant parameters are presented in Table 5.

The defender can move only left or right. It can fire a missile and determine whether it is in the flight path of a bomb. It can also detect whether attackers are to its right or left. The function set contains an if-construct and LISP-type PROGN connectives. The fitness of an evolved strategy is assessed according to how successfully a defender manages to cope with the invading hordes of aliens. Missile launches which miss their targets are penalised according to the distance by which they go wide. If the defender is killed or an attacker manages to land unscathed, then additional heavy penalties are applied. Hence, optimum (zero) fitness can be achieved only if a strategy results in the elimination of all attackers in all 50 randomised scenarios.

4.2.3 Parsing

In this problem, the aim is to evolve programs which are capable of parsing arithmetic and logical expressions. The output of a successful parser is the postfix (Reverse Polish) form of each test expression. Again, full details can be found elsewhere [20], but the relevant parameters are given in Table 6.

The terminals for this problem are used for inspecting and manipulating the input expression items and the operator stack. The function set has an if-construct, a while loop and a progn2 connective. The test data set comprises 30 arithmetic and logical expressions of varying complexity. To prevent programs from becoming stuck in infinite loops, a timeout mechanism is used to terminate programs if they execute more than 2,000 instructions. A successfully evolved solution is therefore one which correctly parses each of the 30 input expressions within this instruction limit.

It will be seen that each of the above problems contains an if-then-else construct; additionally, the maze and parsing problems have a while-loop construct available.

Table 6 Tableau for the parsing problem

Objective	Convert arithmetic and logical expressions to postfix
Terminal set	Item-val, stack-top, operand, lteq, output-item, output-stack, push-item
Function set	If-then-else, while, progn2
Fitness cases	30 Expressions
Fitness	Number of incorrect postfix expressions
Restrictions	Programs timed out after 2,000 instructions

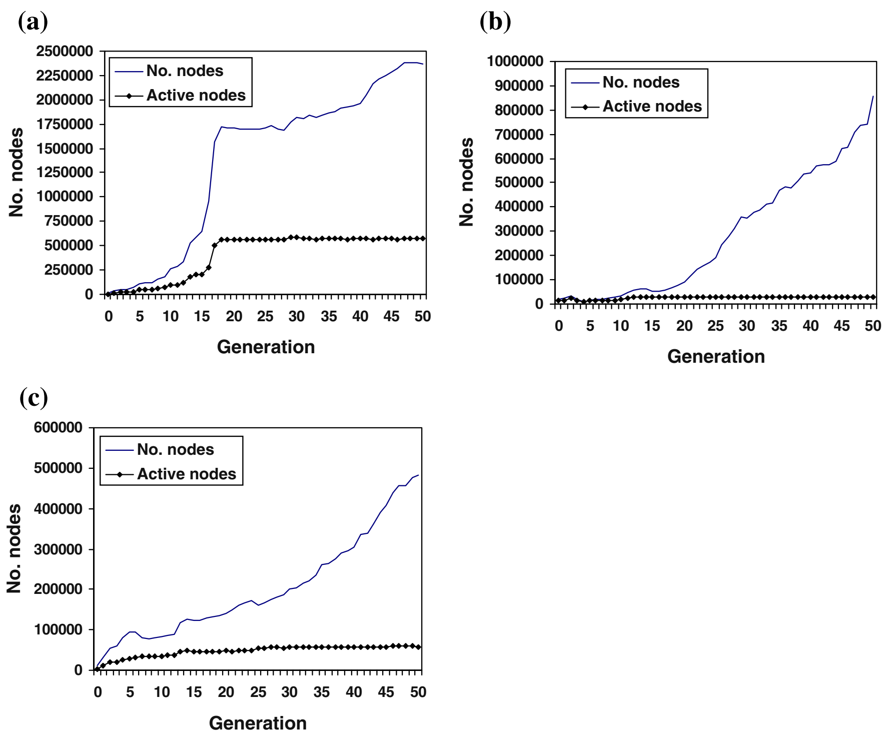


Fig. 7 Extent of dormancy in a typical run of **a** the maze navigation problem; **b** the defence strategy problem; and **c** the parsing problem

There is therefore at least the potential for dormancy to arise in all three. Figure 7 shows how dormancy increases during typical runs of each problem. Although the numbers of nodes involved vary considerably, the shapes of the graphs are remarkably similar to that for the ant problem. Code bloat is evident in all three, but in each case the number of active nodes soon reaches a plateau and stays there until the end of the run.

Figure 8 charts the extent of dormancy in a sequence of 20 runs of each problem, while Fig. 9 shows the percentages of FPCs occurring in the same sequence. The levels of each are considerable, suggesting that the execution savings to be gained via the evaluation avoidance technique described earlier might also be substantial. As before, we can perform comparisons based on the total number of program nodes evaluated, and the elapsed times taken for a sequence of 100 runs. The results are presented in Table 7. The average reduction in the number of nodes evaluated per run is remarkably consistent, both for these problems and the ant problem discussed previously. Moreover, the maximum reduction achieved in any run is, again like the ant problem, above 80% for each problem. Focusing on the elapsed times, the savings obtained by moving to a non-FPC-only approach are 57% in the case of the maze problem, 60% for the defence problem, and 59% for the parsing problem.

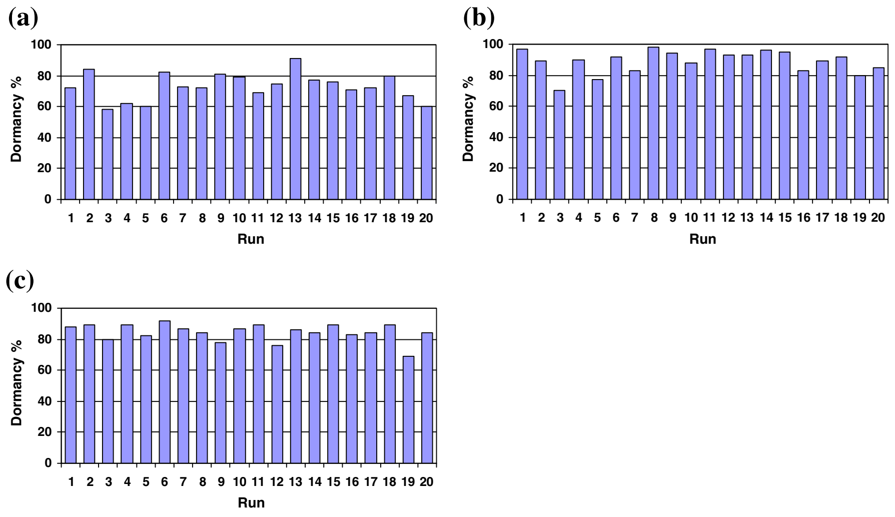


Fig. 8 Final dormancy levels in a sequence of 20 runs of **a** the maze navigation problem; **b** the defence strategy problem; and **c** the parsing problem

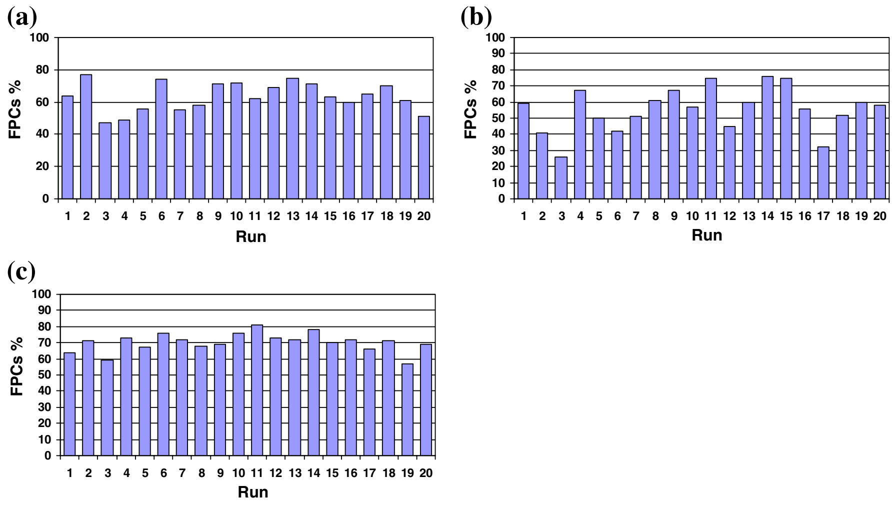


Fig. 9 Percentage of FPCs throughout each of 20 runs of **a** the maze navigation problem; **b** the defence strategy problem; and **c** the parsing problem

4.3 Boolean problems

The technique described in the above sections provides a simple and easily implemented method for minimizing fitness evaluations and hence lowering execution times. It is especially suited to problems which satisfy two conditions: firstly, the computational cost of invoking the fitness function should be relatively

Table 7 Execution efficiency comparisons for the maze, defence and parsing problems

	Maze	Defence	Parsing
Avg. nodes/run (conventional) ($\times 10^6$)	387	193	248
Avg. nodes/run (non-FPC) ($\times 10^6$)	150	102	74
Avg. reduction	63%	64%	65%
Max reduction	81%	87%	85%
Min reduction	37%	10%	45%
Significant?	Yes	Yes	Yes
Time 100 runs (conventional) (s)	1,203	1,422	635
Time 100 runs (non-FPC) (s)	515	578	259

high; and secondly, the function set for the problem should offer the possibility of non-execution of some program nodes (via, say, WHILE or IF functions). With these conditions in mind, the nature of Boolean logic problems is such that they are of particular interest.

Consider, for example, the problem of evolving multiplexer logic. A multiplexer has a number of data inputs and a number of address inputs. The binary value present on the address lines selects one of the data inputs, the value of which is then passed directly onto the single output. In a 6-multiplexer (6-mux), the binary address held on 2 addressing inputs selects one of 4 data inputs to be passed through. In evolving programs to implement a 6-mux, fitness evaluation is often exhaustive: i.e. each program is applied to all 64 possible combinations of the inputs in order to determine its fitness value. More complex multiplexers require correspondingly greater computation time if evaluation is to be exhaustive; for example, an 11-mux, with 3 address and 8 data lines, has 2,048 possible test cases.

The 6-mux problem as it is stated for GP purposes usually has the function and terminal sets given in Table 8. This table also shows the other parameters that we will use in the experiments to be described here.

It will be seen that the function set for this problem contains an IF-statement, and so the potential for dormancy is already present. However, it may be possible to increase the amount of dormancy (and hence the chances of FPCs occurring) still further.

In writing the fitness function for the 6-mux problem, code must be written to evaluate each of the node types in the function and terminal sets. One way of implementing, say, the AND function might be as follows:

Table 8 Tableau for the 6-mux problem

Objective	To evolve a program equivalent in operation to a 6-multiplexer
Terminal set	D0, D1, D2, D3, A0, A1
Function set	AND, OR, NOT, IF
Fitness cases	64, Representing all combinations of inputs
Fitness	Number of mismatches with expected outputs (0–64)

```
case AND_FN:
     return(evaltree() AND evaltree());
```

This code simply calls the evaltree() function to evaluate each of the expression sub-trees forming the two arguments, and then ANDs the results. However, an alternative way of writing the code for this node type is as follows:

```
case AND_FN:
     if (NOT evaltree())
     { skip_arg(); return (0);
     }
     else return (evaltree());
```

In this version, the value of the first operand is determined via a call to evaltree(). If the result of this expression is FALSE (logic 0), then the result of the AND function as a whole must be zero, irrespective of the value of the second argument tree. Hence, the second sub-tree can be skipped over and a zero value returned as the result. Only if the first argument evaluates to TRUE (logic 1) must the second argument be evaluated, as it then determines the value of the final output.

This implementation approach is often referred to as short-circuit evaluation, and it also applies to OR functions and some other forms of Boolean logic. In conventional GP, its advantage lies in the fact that it is sometimes possible to skip evaluation of parts of a program tree, leading to savings in computation time. With regard to the work described in this paper, the approach may offer a further advantage: if some sub-trees are *always* skipped, then those sub-trees can be regarded as dormant, and the chances of fitness-preserving crossovers (FPCs) occurring are increased. The first thing to be determined is how much of an effect short-circuit evaluation can have on the extent of dormancy in a population.

In the graph of Fig. 10, the uppermost line shows how the total number of program nodes in a population changes for a single run of the 6-mux problem. As with the other problems considered so far, the phenomenon of code bloat is clearly

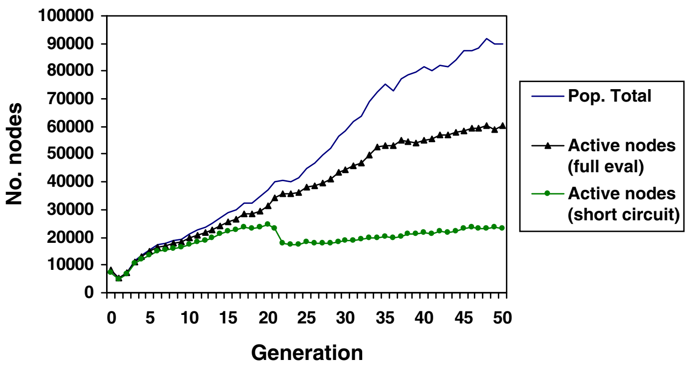


Fig. 10 Total and active nodes in one run of the 6-mux problem

evident as evolution proceeds. The line below that charts the number of nodes which are active (i.e. executed) in the population when Boolean functions are fully evaluated (i.e. both expression sub-trees traversed). The gap between these two lines represents the amount of dormancy, which by generation 50 has reached 33% (one-third) of all nodes. If we now convert our Boolean functions to use short-circuit evaluation, we obtain the bottom line in the graph. The widened gap at generation 50 now represents a level of dormancy that is approximately 75% of all nodes in the population.

The graph of Fig. 11 charts the final levels of dormancy in a sequence of 20 runs of the 6-mux problem. It can be seen that in some cases, such as run 12, converting full evaluation to short-circuit evaluation has little effect on the extent of dormancy in the population. In others, the impact can be dramatic. For example, the introduction of short-circuit evaluation raises the dormancy level from 7 to 67% in run 9, and from 8 to 70% in run 17. A statistical *t*-test on these figures indicates that the increases are significant at the 99.9% confidence level.

Figure 12 shows the percentage of FPCs for the same sequence of 20 runs of the 6-mux problem. Even if full evaluation is used, the number of FPCs is often large, but again the use of short-circuit evaluation can increase this level markedly. In run 9, for example, the number of FPCs performed using full evaluation is, over the whole run, so small as to be indiscernible on the chart. However, moving to

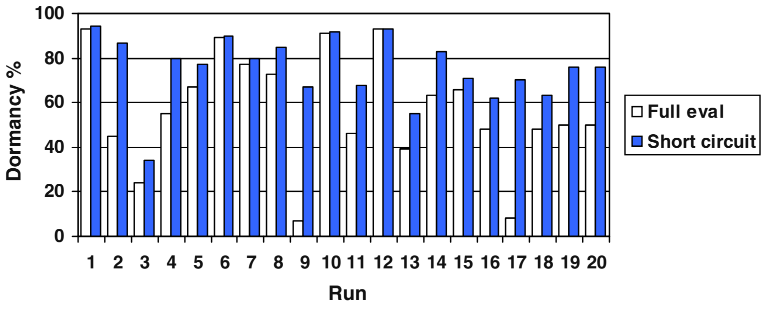


Fig. 11 Final levels of dormancy for full and short-circuit evaluation in the 6-mux problem

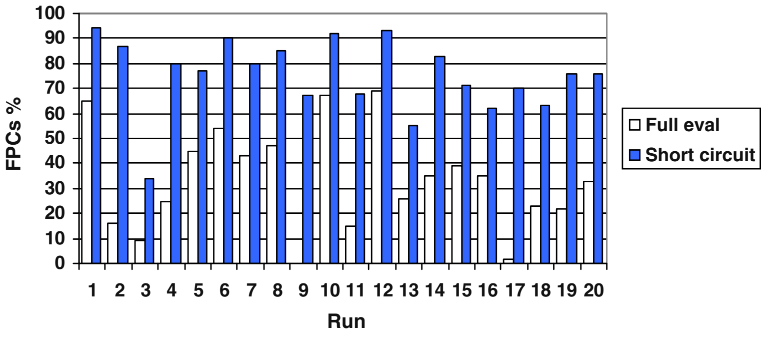


Fig. 12 Percentage of FPCs in 20 successive runs of the 6-mux problem

Table 9 Execution efficiency comparisons for 6-mux problem

	Full evaluation	Short-circuit
Avg. nodes/run (conventional) ($\times 10^6$)	63	31
Avg. nodes/run (non-FPC) ($\times 10^6$)	46	17
Avg. reduction	36%	50%
Max reduction	76%	77%
Min reduction	0.9%	14%
Significant?	Yes	Yes
Time 100 runs (conventional) (s)	124	101
Time 100 runs (non-FPC) (s)	95	57

short-circuit evaluation raises this number to almost 70% of all crossovers in the run. In 14 out of the 20 runs, the level of FPCs obtained using short-circuit evaluation is at least 70%. Again, a t-test shows that these increases are statistically significant at the 99.9% level.

Table 9 compares the various fitness evaluation methods for 100 runs of the 6-mux problem. Conventional GP is compared with our method of minimising fitness evaluations, using both full evaluation of Boolean operands, and short-circuit evaluation.

It will be seen from these figures that the use of short-circuit evaluation alone is worthwhile as a means for improving efficiency. In conventional GP, it reduces by about a half the number of nodes evaluated during execution. However, the savings in execution time are not as great because the number of fitness function calls remains unchanged. Adding in our fitness function avoidance method reduces the number of node evaluations by almost half again, and provides further benefits in terms of the elapsed execution time: the figure of 57 s represents a saving of 54% on the conventional approach using full evaluation, and a saving of 44% on the conventional approach using short-circuit evaluation.

The reduction in execution time for the 6-mux problem derives in part from the conversion of the Boolean evaluation method to a short-circuit mechanism, but also from the fact that the problem as stated contains an IF-statement which may give rise to dormancy. A natural question concerns what will happen for Boolean logic problems which do not make use of IF functions or loops, but contain only traditional gate-type functions such as AND, NAND and OR.

The even-5 parity problem provides one such GP experiment. The idea is to attempt to evolve a program which, given 5 inputs, will return TRUE if the number of inputs set to logic 1 is even, and FALSE otherwise. The function set comprises only the four truth gates AND, OR, NAND, NOR. The parameters for this problem are given in Table 10, and match those used by Koza [17].

If, in a problem such as this, we make use of full evaluation of all arguments to the Boolean functions, then there will of course be no dormancy and hence no FPCs. FPCs arise only when we introduce short-circuit evaluation. The prevalence of FPCs for a sequence of 20 runs of the even-5 parity problem is shown in Fig. 13. It can be seen that the figure tends to hover around the 50% mark.

Table 10 Tableau for the even-5 parity problem

Objective	To evolve a program capable of determining if the number of logic 1 s on the 5 inputs is even
Terminal set	D0, D1, D2, D3, D4
Function set	AND, OR, NAND, NOR
Fitness cases	32, Representing all combinations of inputs
Fitness	Number of mismatches with expected outputs (0–32)
Other parameters	Population size = 4,000; Generations (total offspring/population size) = 101; prob. crossover = 0.9; no mutation; prob. internal node used as crossover point = 0.9

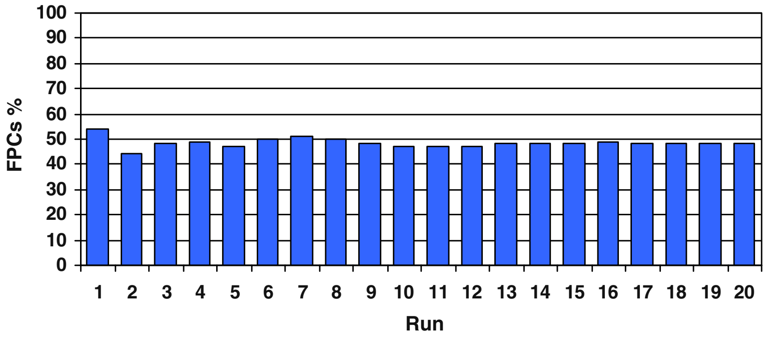


Fig. 13 Percentage of FPCs in 20 runs of the even-5 parity problem. Short-circuit evaluation is in use

Table 11 Efficiency comparisons for the even-5 parity problem

	Full evaluation	Short-circuit
Avg. nodes/run (conventional) ($\times 10^6$)	12,424	955
Avg. nodes/run (non-FPC) ($\times 10^6$)	–	483
Avg. reduction	–	52%
Max reduction	–	66%
Min reduction	–	34%
Significant?	–	Yes
Time 20 runs (conventional) (s)	4,947	811
Time 20 runs (non-FPC) (s)	–	439

Table 11 gives the node evaluation counts and execution times for the even-5 parity problem, using the various methods. Because of the larger population size and doubling of the generations per run, evolution takes much longer, and we have therefore restricted our figures to just 20 runs. Perhaps the most striking figures in this table are those for the conventional approach to fitness determination using full argument evaluation, the huge number of node evaluations per run leading to a substantial execution time. Since all operands are evaluated, there are no dormant nodes and therefore no scope for improving efficiency. As before, however, the introduction of a short-circuit mechanism improves things dramatically, with the fitness evaluation avoidance method adding further substantial savings. In fact, the combination of the two techniques provides a 91% reduction in execution time when compared against the full evaluation approach.

From this we can deduce that even purely Boolean problems (i.e. those for which the function set comprises only Boolean operators) are amenable to efficiency gains via the introduction of our technique for avoiding fitness evaluations. The criterion is that the operators should be capable of being implemented in the form of IF-statements that may have the opportunity of circumventing the execution of one or more of their branches. This criterion is satisfied not only by the binary versions of the operators we have already used in the problems above (AND, NAND, OR, etc.), but also multi-input versions such as 3-input AND-gates etc.

The execution savings achieved depend greatly upon the amount of work that is performed in the fitness evaluator function. In general, as problems scale up, so does the complexity of the fitness function, and avoidance of its execution therefore becomes even more beneficial in improving run-times. To take an example, consider an 11-input multiplexer, in which 3 address lines select one of 8 data inputs. Using a population size of 2,000, the average time for a single run when using full evaluation in a conventional GP system is 163 s. This is largely due to the fact that the fitness evaluator must apply $2^{11} = 2{,}048$ test data cases to each individual. If our fitness evaluation avoidance method is introduced, this time reduces to 138 s. Similarly, if short-circuit evaluation is used, the conventional system takes 146 s, while the FPC-excluding system takes 100 s. These mean times for single runs should be compared with the times for 100 runs of the 6-mux problem as given in Table 9.

5 Performance

So far, we have concentrated on the efficiency gains that can be achieved via the avoidance of fitness evaluations of individuals created by FPCs. A natural question is whether FPCs should be allowed to proceed in the first place, the argument being that, since they create individuals that are functionally identically to one of their parents, they do nothing to widen the search of the program space. This is not to argue that FPCs are entirely worthless, since they may still involve the transport and dissemination of valuable code segments, but it seems worth investigating whether a potentially more extensive search can achieve gains in the numbers of solutions discovered and the effort required to find them.

To make suitable comparisons, we can make use of performance graphs such as those employed by Koza [21]. In these graphs, one line indicates the probability of success in attaining a solution at each generation, whilst another line indicates the number of individuals that must be processed in order to achieve a 99% probability of evolving a solution at each generation. The minimum value of this second line is sometimes referred to as the 'computational effort.'

In Fig. 14 we show the performance graph for the standard artificial ant problem, using the parameters presented earlier. The probability of success $P(M, i)$ is charted for a population size M (500 in our case) at each generation i. $I(M, i, z)$ refers to the number of individuals that must be processed to achieve probability z (=0.99) that a solution will be found at the ith generation. The minimum value, or computational effort E, is indicated by the vertical line drawn at generation 14, and equates to 300,000 individuals. The number of runs required to achieve this value of E is denoted as $R(z) = 40$ runs to generation 14. All of these statistics are gathered over $N = 100$ runs.

Figure 15 shows what happens when FPCs are disallowed. To achieve this, the crossover operation has been altered so that the sub-tree insertion point cannot be at the site of a dormant node: it must be active, and it must therefore lead to the creation of a child with dynamic behaviour that is different from the parent receiving the sub-tree. It can be seen that, although the shape of this graph is slightly different from the preceding one, the general performance characteristics are remarkably similar. The probability of success at generation 50 remains the same at 13%, while the computational effort has in fact increased, but only marginally.

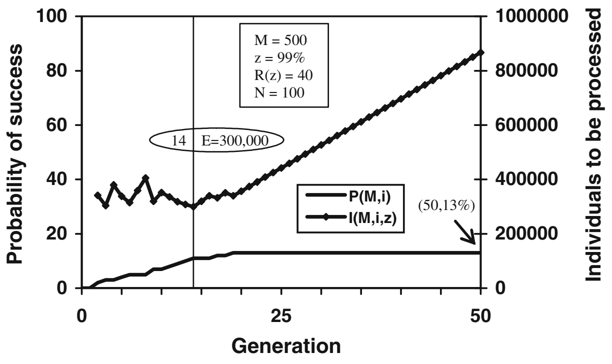


Fig. 14 Performance graph for artificial ant problem using standard GP

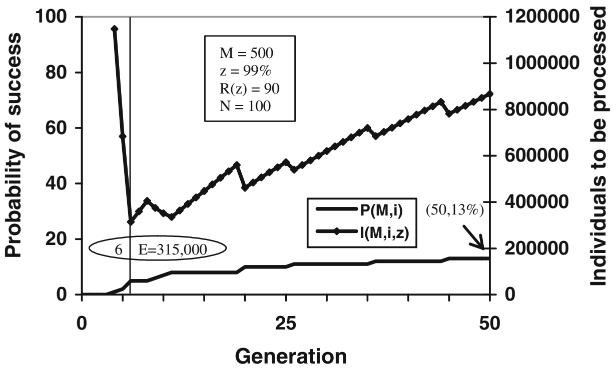


Fig. 15 Performance graph for artificial ant problem when FPCs are prevented

Table 12 Performance comparisons for problems with and without FPCs

Problem	P (success)	Comp. effort
Ant	13	300,000
Ant, FPCs prevented	13	315,000
Maze	35	315,000
Maze, FPCs prevented	79	70,000
Defence	56	56,000
Defence, FPCs prevented	70	36,000
Parse	22	440,000
Parse, FPCs prevented	50	103,500
6-Mux	68	44,000
6-Mux, FPCs prevented	65	38,500
Even-4	14	700,000
Even-4, FPCs prevented	17	490,000

Hence, for this particular problem, the scope for wider exploration of the program space seems to have had little impact on evolutionary performance. It is worth noting that Blickle and Thiele [16] carried out a similar experiment for the ant problem, and that they too found no significant differences between the two approaches; however, they did not consider the range of problems we shall consider, nor perform the same in-depth analysis.

Rather than present a whole series of graphs, Table 12 summarises the performance characteristics for each of the problems we have used in this paper. A minor exception is that for the parity problem we have chosen to use 4 inputs rather than 5. This is not merely because the even-5 problem has very lengthy run times: it also results in an extremely low solution count, making comparisons difficult. Moving to 4 inputs allows us to drop the population size to 500 and the number of generations to 50, consistent with the other problems whilst still generating a sufficient number of solutions to make comparisons meaningful. All other parameters for the problem remain unchanged. For each problem in the table, the figures for the standard approach appear above that for the approach in which FPCs are disallowed.

It will be seen that, as with the ant problem, the two Boolean problems (6-mux and even-4) show very little difference between the two approaches. In contrast, the maze and parsing problems show more than a two-fold increase in performance when FPCs are disallowed, and the defence problem also exhibits a substantial improvement. It is worth remarking that Blickle and Thiele also performed this experiment with the 6-mux problem, and reported a two-fold gain in performance for that problem too. However, the explanation for this seems due less to the benefits of a non-FPC approach and more to the poor showing of their conventional GP approach, which for some reason achieved substantially weaker performance results than our own.

Why should some problems experience a performance gain, and not others? A comparison of the dormancy graphs reveals that the three biggest performance beneficiaries—maze, parsing and defence—are also the leaders in terms of the total number of nodes contained in the population at generation 50, even though the numbers of individuals remains constant. When FPCs are prevented, the dynamic behaviour of children must differ from the parents (even if the overall fitness does not), and it is commonly recognized that crossover is a generally destructive operation. On this evidence alone, a plausible reason for the performance difference is that the large trees appearing in the maze, parsing and defence problems are more resistant to the deleterious effects of crossover, whilst the smaller trees appearing in the other problems are more heavily disrupted. Larger trees contain sub-trees of greater depth, and experimental evidence suggests that evolutionary operators applied to deep sub-trees have less impact on fitness [22]. However, it should be borne in mind that the graphs under consideration here are for single runs only; to reach more general conclusions regarding the observed differences in performance we need to move on to a much deeper analysis of problem behaviour.

Table 13 presents a number of statistics relating to each of our problems. For ease of use, column 2 repeats the probability of success figures previously given in Table 12. In column 3 we present the sizes of individuals selected to mate in crossover operations, averaged over 100 runs. On the whole, this column tends to support the notion that problem-solving performance is related to the sizes of trees involved in crossover. The ant problem, for example, has the smallest parent size of all in its FPC-free version. It also experiences a huge decrease in parent size when moving from the standard version of the problem to that in which FPCs are disallowed. It seems possible, therefore, that any improvement in the scope of the search of the program space may be counteracted by the greater relative disruption of the program trees. In contrast, the maze problem involves the crossover of parents that are very much bigger than those in the other problems, and so the potential for disruption is lessened. Similarly for the parsing problem, in which the average parent size for the FPC-free version is the next biggest after maze.

Table 13 Execution statistics

Problem	P (succ)	Av. parent size	Destructive crossovers (%)	Av. fitness shift	Improving crossovers (%)	Av. improvement
Ant	13	217	33.8	−0.13	0.8	0.06
Ant, FPCs prevented	13	71	80.4	−0.33	1.3	0.06
Maze	35	2,574	24.6	−0.03	1.3	0.02
Maze, FPCs prevented	79	2,962	70.1	−0.08	5.9	0.02
Defence	56	330	30.8	−0.1	1.1	0.08
Defence, FPCs prevented	70	93	61.2	−0.197	2.5	0.08
Parse	22	330	18.0	−0.05	0.1	0.08
Parse, FPCs prevented	50	408	38.2	−0.1	0.5	0.08
6-Mux	68	178	27.2	−0.03	1.5	0.04
6-Mux, FPCs prevented	65	161	33.5	−0.03	1.6	0.04
Even-4	14	299	28.0	−0.03	0.2	0.06
Even-4, FPCs prevented	17	253	37.5	−0.04	0.2	0.06

The even-4 and 6-mux problems also involve the selection of parents which are significantly larger than those in the FPC-free version of the ant problem, and so one might expect a greater improvement for these problems too. However, attempting to compare across problems in this way may not be so straightforward. Unlike the ant problem, the even-4 and 6-mux problems show very little change in average parent size when moving from the standard to the FPC-free approach, and so it may be the case that the amount of disruption caused by crossover remains fairly constant.

More difficult to explain away are the results obtained for the defence strategy problem. The change in average parent size for this problem is similar to that obtained in the ant problem, and yet the defence problem manages to attain a significant improvement in performance.

To analyse this further, we need to discover exactly how much disruption is being caused by the crossover operations in each problem, and whether this is in fact related to tree size. To do so, we will define a destructive crossover operation to be one in which the offspring has a fitness value that is worse than that of the parent receiving the donated sub-tree (the mother). Of course, some crossovers are more destructive than others, so we also need a metric for the amount of disruption caused. To do this, we introduce the notion of *fitness shift*, defined in such a way as to be comparable across problems.

Definition: Let the child program's fitness be C, and its mother's fitness be M. Let the worst possible fitness for the given problem be W (assuming that a fitness value of zero indicates a perfect solution). Then we define the *fitness shift* F for a given crossover operation to be $F = (M - \mathrm{C})/\mathrm{W}$.

Suppose, for example, that the fitness range for a given problem is 0–100, with 0 indicating a solution. If, via crossover, a maternal parent with fitness value 100 (a wholly unfit mother!) were to generate an offspring with perfect (zero) fitness, then that would denote a fitness shift of exactly 1.0. Similarly, a perfect solution that produced a wholly unfit child by crossover would denote a fitness shift of -1.0. All other crossovers can be tagged with a fitness shift somewhere between these two extremes, with negative values indicating destructive crossovers and positive values indicating improving crossovers.

Also contained in Table 13 is the percentage of destructive crossovers and the average fitness shift of all crossover operations, taken over 100 runs of each problem. The worsening of these figures when moving from standard GP to an FPC-free version is to be expected, since we are replacing the neutral FPCs by fitness-altering crossovers which are known to be mostly destructive. In 6-mux and even-4 (which showed little change in performance) the increase in destructive crossovers is not huge, and the fitness shift barely alters. Conversely, the ant problem exhibits a huge leap in the number of destructive crossovers and a corresponding worsening of the average fitness shift. This is supportive of the notion that smaller parents suffer most from crossover.

Again, however, these figures do not offer a full explanation as to why some problems perform better than others when FPCs are eliminated. The three problems which attain the greatest performance boost all exhibit large increases in the percentage of destructive crossovers and in the degradation of the fitness shift. In the case of the maze problem these figures are almost tripled, despite the large tree sizes.

To help explain this, Table 13 also shows the percentage of improving crossovers. It should be noted that, in the FPC-free approach, this figure is not equal to 100 minus the percentage of destructive crossovers; this is because even though crossovers at dormant sites are prevented, there will still be a significant proportion of crossovers which result in a child with fitness equivalent to that of the mother. The final column in the table gives the average fitness shift over these improving crossovers alone. These figures therefore tell us not only how many improving crossovers there are, but also the extent of their contribution to improving overall fitness.

It will be seen that in all cases the average fitness shift does not alter in moving from the standard crossover system to the FPC-free version, and also that these average improvements are all fairly small (<10% of the fitness range). The number of improving crossovers is also very much smaller than the number of destructive crossovers. The figures therefore support the widely accepted notion that crossover is an inherently destructive mechanism.

In terms of explaining the performance results, however, the key thing to note here seems to be the change in the percentage of improving crossovers. For 6-mux and even-4, which showed little in the way of performance enhancement when moving from the standard to the FPC-free approach, there is also little or no change in the number of improving crossovers. For the two problems which achieved the greatest performance boost (maze and parse) there is roughly a five-fold increase in the percentage of improving crossovers. For the next-best performer (defence), there is more than a two-fold increase. The evidence therefore tends to suggest that, even in the presence of large numbers of destructive crossovers, a small increase in the number of improving crossovers can radically enhance performance.

The ant problem requires a little more explanation: the change in the percentage of improving crossovers is only small in absolute terms, but it still represents more than a 50% increase, and it might be wondered why this problem does not fare better when FPCs are disallowed. The answer, it is suggested, is that the positive effects of this increase are counteracted by the massive leap in the number of destructive crossovers. Indeed, it may well be the case that the increase in improving crossovers is the only thing which prevents the FPC-free version of the ant problem from being so very much worse than the standard version.

6 Simplification

The rapid increases in the size and complexity of population members during GP runs means that evolved programs are often extremely difficult to understand. In particular, programs that are flagged as ‘solutions’ (in that they pass all the tests embodied within a GP fitness evaluator) are often not very amenable to subsequent analysis and verification. This approach to generating products which appear to work but which cannot be understood can have the effect of lowering confidence in the suitability of GP as a solution-finding tool.

Simplicity of evolved programs is therefore highly desirable, but it is often overlooked as an objective of GP systems. A nod towards it can be achieved by defining the 'best' program in a population to be the shortest of those programs with the highest fitness. This can be useful if it is only the fittest programs that are of interest, but it does nothing physically to reduce the number of tree nodes in the population.

Other techniques take more direct action. For example, program trees can be constrained to be less than a certain size or depth. However, limiting the program search space in this way can reduce the effectiveness of a GP system at finding solutions. More sophisticated approaches involve altering the fitness function so that it penalises excessively long or complex population members by lowering their fitness scores. This 'parsimony pressure' has to be applied judiciously, else it can again be highly constraining on the GP system's freedom to explore the program search space [23].

To avoid altering the nature of the evolutionary process, code simplification is usually done (if at all) as a post-evolutionary event, i.e. at the end of a run. In its usual form, it involves the analysis of the program tree to determine whether sub-trees can be replaced by simpler but functionally equivalent structures, and can be performed either manually or automatically. To achieve such editing, a set of replacement rules must be derived in advance. For example, one rule might be:

$$\mathrm{mult}(\alpha, 0) \rightarrow 0$$

meaning that the multiplication of any sub-tree by zero can be replaced by a zero node.

There are several disadvantages to such an approach. The first is that it is problem-dependent. A new set of replacement rules must be derived for each and every problem. Moreover, if the editing is to be performed automatically, then significant implementation work may be required each time. The effort this entails depends not only on the nature of the problem, but also on how extensive the rule matching is to be. For example, consider the replacement rule

$$\mathrm{if}(\alpha, \beta, \beta) \rightarrow \beta$$

which states that the entire if-statement can be replaced if its then-part is equivalent to its else-part. Based on this, it is easy to see that a sub-tree such as if(a0, d1, d1) can be replaced by d1. However, it would be difficult for an automated system to deal with more complex arguments to the if-statement. Even relatively simple constructs such as if(a0, and(d0, d1), and(d1, d0)) require additional checking to determine that the then-part and the else-part are functionally identical albeit structurally different.

Another problem with this editing approach is that it is based on a *static* analysis of the program code. Replacement decisions have to be made with respect to the problem domain, i.e. they are based on what could possibly occur rather than what actually does occur during the evolutionary run. The consequence is that replacement decisions are often overly conservative. An example is

while(wall_ahead, turn_left)

which in itself appears not to be capable of simplification. However, perhaps because of the nature of the maze being navigated or the behaviour of the preceding code, it may be the case that wall_ahead is never true when this loop is reached, in which case it could be deleted. The problem is that static analysis may not be able to uncover this characteristic; only dynamically gathered information can tell us whether the loop does in fact make any contribution to the fitness of the individual.

A third difficulty with static post-evolutionary simplification is that what seem to be obvious replacement strategies are sometimes complicated by the need to consider side-effects of functions and terminals. In the Santa Fe artificial ant problem it might seem at first sight that a sub-tree such as

$$\text{PROGN2}(\text{LEFT}, \text{PROGN3}(\text{LEFT, LEFT, LEFT}))$$

could be deleted, since it merely causes the ant to execute four left turns, leaving it exactly as it started. However, this overlooks the fact that in the ant problem each movement consumes a unit of time, and so deleting this construct would alter the program's timing. The only justification for removing it would be if it were never executed at all, but again ascertaining this will usually require dynamic monitoring.

In previous sections of this paper we have outlined a method for gathering information about individuals during the execution runs of a GP system. This tells us exactly which nodes of an individual's program tree were never executed in determining its fitness. It follows from this that an obvious and easily-implemented simplification algorithm is to replace the root node of each dormant sub-tree with a single node from the terminal set (it doesn't matter which, since it will not be executed) [24]. The prevalence of dormancy as discussed in earlier sections suggests that this will immediately lead to dramatic simplification of many of the individuals in a population.

Since it is based on dynamic information, the technique does not suffer from many of the problems associated with static approaches: there is no need to worry about the side effects of, or to be wary of deleting, code that is never activated; it can perform simplifications that are impossible or costly to detect statically; it is problem-independent, and so requires no re-implementation effort for each problem; and it re-uses information already gathered via the visit tree, making it cheap to put in place and run.

However, what we wish to explore further in this section are the ways in which the simplification algorithm can be extended beyond mere collapse of dormant sub-trees. We shall do this using the example of the 6-multiplexer. It will be recalled that this problem makes use of a function set comprising {AND, OR, NOT, IF} and a terminal set {D0, D1, D2, D3, A0, A1}. In a solution, the two-bit address on A0 and A1 specifies which of the four data inputs D0-D3 is passed onto the output.

Consider now the sub-tree AND(α, β) in an evolved program, where α and β are argument sub-trees, and suppose that the visit tree tells us that β is dormant. As we have already discussed, a possible simplification would be to replace β by an arbitrary terminal. However, this neglects a broader implication of the dormancy: If β has never been visited, then that can only be because α has always evaluated to zero; and if α is always zero, then the value returned by the AND operation must always be zero. Hence, we can do better than simply replacing β: by introducing a ZERO token we can replace the whole AND(α, β) sub-tree. Note that this new ZERO token is purely a notational convenience for the purposes of presenting programs in a more simplified form after evolution has taken place; it is not being suggested that it become incorporated into the terminal set and used during the evolutionary process.

Similar reasoning applies to the OR function. If the second argument sub-tree is dormant, then it must be because the first argument always evaluates to logic one, and so the whole OR-tree can be replaced by a ONE token.

For the IF function, we can begin by examining the visit tree corresponding to the then-part (the second argument). If that sub-tree is dormant, then that must be because the predicate forming the first argument always evaluates to FALSE, and so we can replace the entire IF-tree by the sub-tree forming the else-part (third argument). If, on the other hand, the then-part is active but the else-part is dormant, then we can replace the entire IF-tree by the then-part. Note that irrespective of any simplifications applied to the IF-tree as a whole, further simplifications may be possible for each of its three argument sub-trees.

Application of this algorithm to a program leaves it in a simplified form perhaps containing some ZERO and ONE nodes. However, this is still not necessarily the end of the simplification process. If either of the arguments of an AND function turns out via simplification to be ZERO, then the whole sub-tree must be ZERO; if either argument is ONE, then the value of the function is simply the sub-tree forming the other argument. Similarly, if either of the arguments of an OR function is ONE, the value of the function as a whole must be ONE; if either argument is ZERO, the function can be replaced by the other argument sub-tree. Finally, a NOT node can of course be replaced if its argument simplifies to ONE or ZERO. Each replacement action introduces the possibility for further substitutions, and so repeated passes of this second stage of simplification can remove most if not all of the temporary ZERO and ONE tokens that have been introduced.

Figure 16 shows the pseudocode form of the algorithms we have described. The tree to be simplified is passed to the *simplify* function. The *phase1* function recursively collapses sub-trees into ZERO and ONE values where possible. The *phase2* recursive function then attempts to make additional simplifications by removing as many of these ZERO and ONE constants as it can. This second stage is repeated until no further reductions are possible.

An example may serve to illustrate this two-stage process. Figure 17 shows a tree that appeared early in one run of our GP system on the 6-multiplexer problem. This is not a huge tree, but a fair amount of thought would be required to determine how to simplify it by hand. Examination of our visit tree, however, reveals that there are three nodes that are dormant; these are denoted by the dashed rectangles in the diagram. The trivial strategy of replacing dormant root nodes by arbitrary terminals is clearly not going to have any effect in reducing the size of this tree, but consider what happens when we apply the first phase of our simplifier algorithm. The way that this has been implemented is as a recursive algorithm applied initially to the root node of the tree. When the sub-tree labelled S1 in the diagram is reached, the simplifier detects that the D0 node is dormant, and so replaces the whole of S1 by a logic ONE token, as described earlier. Similarly, when S2 is reached, it finds that D3 is dormant and replaces the whole of S2 by a ZERO token. This gives us the simplified tree shown in Fig. 18.

Fig. 16 Pseudocode for program simplification. To avoid confusion with node names, the logical AND operator is represented using the && symbol

```
phase1(node)
// Recursive function to replace dormant nodes with
// ZERO and ONE literals where possible
{
   // Handle those cases that can be simplified
   if (node is AND && argument 2 is dormant)
      transmit(ZERO);
   else if (node is OR && argument 2 is dormant)
      transmit(ONE);
   else if (node is IF && then-part is dormant)
      // Only else-part is needed
      phase1(else-part);
   else if (node is IF && else-part is dormant)
      // Only then-part is needed
      phase1(then-part);
   else
   // Can't simplify, so output node and try sub-trees
   {  transmit(node);
      for each argument sub-tree of node
         phase1(argument);
   }
}


phase2(node)
// Recursive function to simplify expressions
// containing ZERO and ONE constants
{  if (node is AND && either argument is ZERO)
      transmit(ZERO);
   else if (node is AND && either argument is ONE)
      phase2(other argument);
   else if (node is OR && either argument is ONE)
      transmit(ONE);
   else if (node is OR && either argument is ZERO)
      phase2(other argument);
   else if (node is NOT && argument is ZERO)
      transmit(ONE);
   else if (node is NOT && argument is ONE)
      transmit(ZERO);
   else
   // Can't do anything. Output node and try arguments
   {  transmit(node);
      for each argument sub-tree of node
         phase2(argument);
   }
}


simplify(tree)
// Simplify individual's code tree
{  phase1(tree);
   do  phase2(tree);
   while (tree has changed);
}
```

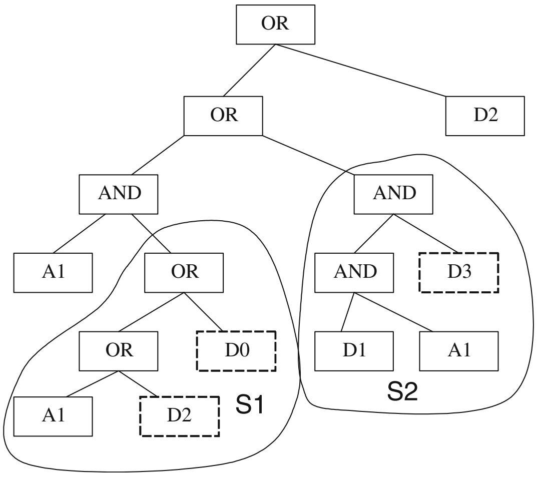


Fig. 17 Example program to be simplified. This individual occurred during a run of the 6-mux problem

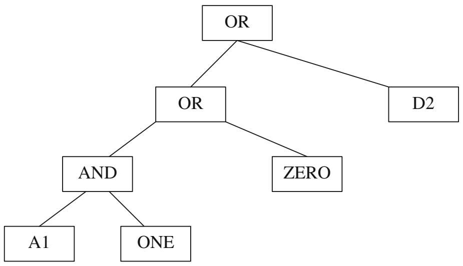


Fig. 18 Simplified program from 6-mux run, with S1 and S2 replaced by logic ONE and ZERO, respectively

It is now easier to understand why the three dormant nodes were never visited in the first place. Since it is the rightmost argument of an AND function, sub-tree S1 is visited only when A1 has the value one. If A1 is one, then the value of OR(A1,D2) must be one, and this can be determined without seeking the value of D2. Further, if OR(A1,D2) is one, then the value of D0 in S1 is redundant, and it can remain dormant. It then follows from this that S2 is visited only when A1 is zero; when that is the case, the leftmost argument of the root AND function of S2 is zero, and the value of D3 is not required.

The tree shown in Fig. 18 is that obtained following phase one of our simplification. In phase two, we make multiple recursive passes over the tree, looking for opportunities to make further replacements. Firstly, AND(A1, 1) can be replaced by A1; following that, OR(A1,0) can be replaced by A1. The final simplified tree is shown in Fig. 19. It represents a reduction from 15 nodes down to just three—an 80% saving in size.

Figure 20 illustrates another simplification example, this time containing an if-statement. The visit tree tells us that the then-part of the IF-statement is never traversed. This is because the root OR function needs to know the value of the IF-function only when D0 is false (logic zero). Hence, the whole IF sub-tree can be replaced by its else-argument, giving the minimised tree shown in Figure 20b.

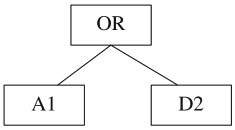


Fig. 19 Final simplified form of program from 6-mux run

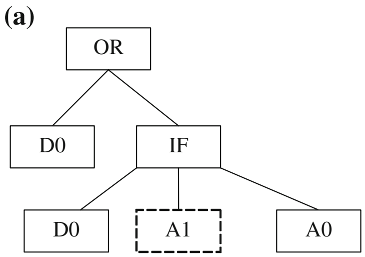


Fig. 20 Another example program from 6-mux run, **a** before simplification; and **b** after simplification

Table 14 Simplification statistics for the 6-mux problem

Simplification coverage	Maximum reduction (%)	Average reduction (%)	Significant?
Solutions only, with early termination	97	40	Yes
All programs, after 50 generations	99.5	85	Yes

Table 14 shows the effects of our simplification algorithm over 100 runs of the 6-multiplexer problem. As in all our previous experiments, significance is tested using a t-test at the 99.9% confidence level. The first row deals only with programs that represent complete solutions to the problem, since they are usually of more interest than partially correct programs. Each run was terminated as soon as a solution was generated. It can be seen that the biggest reduction was 97%; this was for a solution that was cut down from 716 to 20 nodes. On average, the algorithm reduced solution size by 40%.

The second row of the table shows what happens when each run is allowed to proceed for the full 50 generations, following which all population members are simplified. The maximum reduction achieved was 99.5%, for a program that was condensed from 2,596 nodes down to just 12 nodes. On average, population members were reduced in size by 85%.

7 Conclusions

In this paper, we have considered a particular subset of introns that arise during runs of GP problems. Specifically, we have focused on those nodes of a program code tree which are never executed for any of the test cases applied during fitness evaluation. We refer to such nodes as *dormant*, whilst those nodes which are executed are termed *active*. The identification and location of dormant nodes does not require computationally expensive analysis of the code; it can be gleaned purely from information gathered dynamically during execution. The mechanism for achieving this is simple and easily implemented in a GP system.

Experimentation reveals that the presence of dormancy in a population is extensive across a range of problems, and we have investigated three ways in which this phenomenon can be exploited. The first of these concerns the execution efficiency of a GP system when evolving solutions to these problems. Whenever an individual is created via crossover at the site of a dormant node, the fitness of that individual cannot differ from that of its mother, and so the invocation of the fitness function for that program can be avoided. Experimentation shows that the occurrence of such FPCs is frequent, and that the consequent reduction in the number of fitness evaluations can lead to substantial lowering of execution times. For Boolean logic problems in particular, the use of so-called ‘short-circuit’ evaluation can increase dormancy still further, thereby providing additional efficiency gains.

It should be noted that nothing need be sacrificed to achieve these gains. The evolutionary process itself remains unaffected. In other words, the population evolves in exactly the same way, leading to identical solutions produced at identical points in the process. The only difference is that it all happens much faster. For the approach to be suitable for a given problem, however, one of the requirements (others, such as a fixed test data set, are discussed later) is that fitness evaluation be relatively expensive to perform. Avoidance of fitness evaluations which execute quickly will not be sufficient to counteract the overheads which the approach introduces. On the other hand, problems which are more computationally intensive than those which we have used in this paper may bring even greater efficiency gains (e.g. higher-order parity problems). Furthermore, since it is clear that the percentage of dormant code increases during a run, the approach is likely to have greater beneficial impact as users of GP systems tackle ever more complex problems requiring larger populations running over more generations.

The second way in which dormant code can be exploited concerns the effectiveness of a GP system in finding solutions to problems. The idea is that instead of allowing FPCs to proceed, they are identified and eliminated in the hope that a more extensive search of the program space will encourage diversity and provide a greater chance of finding solutions. The experiments we performed indicate that this approach works for some problems but not for others. Further investigation suggests that the decisive factor may be a change in the number of improving crossovers occurring. However, this is a determination that can only be made *following* a GP run; it is not known whether there are other characteristics of a given problem that will allow the determination to be made prior to execution, and this may be a possible avenue for future work.

Thirdly, we have examined ways in which the known presence of dormancy can be used to aid in the simplification of evolved programs. The techniques we have described go beyond the mere replacement of dormant sub-trees by single arbitrary nodes. Even though the reductions in tree sizes achieved are not guaranteed to be maximal, experimentation shows that they are certainly substantial. Because the simplification method works by replacing the sub-tree *containing* the dormant code, rather than replacing the dormant code with a simpler unexecuted node, *all* dormancy is removed from the program tree.

It should be pointed out that success of these three forms of exploitation for a given problem relies on the existence of certain properties of that problem. Firstly, its function set must allow at least the potential for the non-execution of some program tree nodes. This may entail the presence of, say, an if-construct or while-loop, in which the possibility exists that one or more paths through the program are never pursued over the range of test data inputs. For Boolean problems, it may involve the re-writing of some of the function algorithms so that they are made equivalent to if-constructs. Encouragingly, the experiments suggest that the presence of only one such construct is sufficient to give rise to a level of dormancy that is significant enough to be exploitable in the ways we have outlined.

The other property that a problem must possess is that its test data set remain fixed for each invocation of the fitness function. This does not necessarily mean that the test set has to be known in advance of execution—it may still be produced by, say, a random number generator (as we did in our maze and defence problems)—but once produced it must not vary each time an individual is evaluated. The definition of dormancy is with respect to a given data set; if these values are allowed to vary, then previously unexecuted paths may become active.

That said, both of these restrictions give rise to other possibilities for future work. We are interested, for example, in the range of problems to which the approach described here is applicable. In particular, symbolic regression problems, which usually contain neither alternation or iteration, are currently under investigation. Another path of research concerns those problems in which the data set does not remain fixed, where interesting questions are whether it is still possible to mark some nodes as dormant, and whether this requires some foreknowledge about the possible test inputs.

References

1. T. Soule, in *Proceedings of EuroGP 2002, LNCS 2278*, ed. by J.A. Foster et al. Exons and code growth in genetic programming, (Springer, Berlin, Heidelberg, 2002), pp. 142–151
2. T. Soule, J.A. Foster, J. Dickinson, in *Genetic Programming 1996: Proceedings of the First Annual Conference*, ed. by J.R. Koza et al. Code growth in genetic programming, (MIT Press, Cambridge MA, 1996), pp. 215–223
3. T. Soule, *Code Growth in Genetic Programming*. PhD Thesis, University of Idaho, 1998
4. S. Luke, in *Late Breaking Papers, Proceedings of Genetic and Evolutionary Computation Conference (GECCO)*, ed. by D. Whitley. Code growth is not caused by introns, (Las Vegas, USA, 2000), pp. 228–235
5. J. Miller, in *Late Breaking Papers, Proceedings Genetic and Evolutionary Computation Conference (GECCO)*, ed. by E.D. Goodman. What bloat? Cartesian genetic programming on boolean problems, (San Francisco, CA, USA, 2001), pp. 295–302
6. P. Smith, K. Harries, Code growth, explicitly defined introns and alternative selection schemes. Evolutionary Computation **6**(4), 339–360 (1998)
7. J. Stevens, R. B. Heckendorn, T. Soule, in *Proceedings of Genetic and Evolutionary Computation Conference (GECCO)*, ed. by H.-G. Beyer, U.-M. O'Reilly. Exploiting disruption aversion to control code bloat, (ACM Press, Washington, DC, New York, 2005), pp. 1605–1612

8. S. Mahler, D. Robilliard, C. Fonlipt, in *Proceedings of EuroGP 2005, LNCS 3447.* Tarpeian bloat control and generalization accuracy, (Springer, Berlin, Heidelberg, 2005), pp. 203–214
9. R. Poli, in *Proceedings of EuroGP 2003, LNCS 2610.* A simple but theoretically-motivated method to control bloat in genetic programming, (Springer, Berlin, Heidelberg, 2003), pp. 204–217
10. S. Luke, L. Panait, A comparison of bloat control methods for genetic programming. Evolutionary Computation **14**(3), 309–344 (2006)
11. P. Nordin, F. Francone, W. Banzhaf, Explicitly defined introns and destructive crossover in genetic programming, in *Advances in Genetic Programming*, vol. 2, ed. by P.J. Angeline, K.E. Kinnear (MIT Press, Cambridge, MA, 1996), pp. 111–134
12. S. Carbajal, F. Martinez, in *Genetic Programming and Evolvable Machines*. Evolutive introns: a non-costly method of using introns in GP, vol. 2, no. 2, June 2001, pp. 111–122
13. H. Iba, M. Terao, in *Proceedings of Genetic and Evolutionary Computation Conference (GECCO)*, ed. by D. Whitley. Controlling effective introns for multi-agent learning by genetic programming, (Las Vegas, USA, 2000), pp. 419–426
14. W.B. Langdon, T. Soule, R. Poli, J.A. Foster, The evolution of size and shape, in *Advances in Genetic Programming*, vol. 3, ed. by L. Spector, et al. (MIT Press, Cambridge, MA, 1999), pp. 163–190
15. T. Soule, J.A. Foster, in *Proceedings 1998 IEEE International Conference on Evolutionary Computation*. Removal bias: a new cause of code growth in tree based evolutionary programming, (IEEE Press, Anchorage, Alaska, USA, 1998), pp. 781–786
16. T. Blickle, L. Thiele, in *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94)*, ed. by J. Hopf. Genetic programming and redundancy, (Saarbrücken, 1994), pp. 33–38
17. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (MIT Press, Cambridge, MA, 1992)
18. W.B. Langdon, Data structures and genetic programming, in *Advances in Genetic Programming*, vol. 2, ed. by P.J. Angeline, K.E. Kinnear (MIT Press, Cambridge, MA, 1996), pp. 395–414
19. D. Jackson, in *Proceedings EuroGP 2005, LNCS 3447*, ed. by M. Keijzer et al. Evolving defence strategies by genetic programming, (Springer, Berlin, Heidelberg, 2005), pp. 281–290
20. D. Jackson, in *Proceedings Genetic and Evolutionary Computation Conference (GECCO)*, ed. by H.-G Beyer, U.-M O'Reilly. Parsing and translation of expressions by genetic programming, (ACM Press, Washington, DC, New York, 2005), pp. 1681–1688
21. J.R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs* (MIT Press, Cambridge, MA, 1994)
22. L. Igel, K. Chellapilla, in *Proceedings of Genetic and Evolutionary Computation Conference (GECCO)*, ed. by W. Banzhaf et al. Investigating the influence of depth and degree of genotypic change on fitness in genetic programming, (Morgan Kaufmann, Orlando, Fl, 1999), pp. 1061–1068
23. T. Soule, J.A. Foster, Effects of code growth and parsimony pressure on populations in genetic programming. Evolutionary Computation **6**(4), 293–309 (1998)
24. T. Blickle, in *Proceedings of PPSN IV, LNCS 1141*. Evolving compact solutions in genetic programming: A case study, (Springer, Berlin, Heidelberg, 1996), pp. 564–573