Genet Program Evolvable Mach (2009) 10:307–337
DOI 10.1007/s10710-009-9082-5

ORIGINAL PAPER

Semantic analysis of program initialisation in genetic programming

Lawrence Beadle · Colin G. Johnson

Received: 30 July 2008 / Published online: 7 March 2009

Abstract Population initialisation in genetic programming is both easy, because random combinations of syntax can be generated straightforwardly, and hard, because these random combinations of syntax do not always produce random and diverse program behaviours. In this paper we perform analyses of behavioural diversity, the size and shape of starting populations, the effects of purely semantic program initialisation and the importance of tree shape in the context of program initialisation. To achieve this, we create four different algorithms, in addition to using the traditional ramped half and half technique, applied to seven genetic programming problems. We present results to show that varying the choice and design of program initialisation can dramatically influence the performance of genetic programming. In particular, program behaviour and evolvable tree shape can have dramatic effects on the performance of genetic programming. The four algorithms we present have different rates of success on different problems.

Keywords Genetic programming · Program initialisation · Program semantics · Program structure

1 Introduction

The ultimate goal of this paper is to attain an advanced understanding of the issues involved with program initialisation in genetic programming (GP) through the analysis of different aspects of program initialisation. Although we present four

L. Beadle (✉) · C. G. Johnson
Computing Laboratory, University of Kent, Canterbury CT2 7NF, UK
e-mail: l.beadle-276@kent.ac.uk

C. G. Johnson
e-mail: c.g.johnson@kent.ac.uk

different program initialisation algorithms with varying levels of success, our focus is primarily on testing theories rather than presenting algorithms for practical use.

The ideal initialisation of random programs in GP presents a unique challenge when contrasted with population initialisation for other evolutionary algorithms. Unlike other forms of evolutionary computation, GP relies on the execution (or interpreted execution) of programs in order to attain fitness values (although some work has included program structure as a factor [1–3]). In terms of creating random programs to seed a GP run, the fact that fitness is based on the execution of the program means that we should investigate a semantically diverse starting population, rather than one that is syntactically diverse. It seems reasonable that this would increase the search power of GP.

In order to test the theory that increased semantic diversity affects GP results, we present an entirely semantically driven initialisation (SDI) algorithm (built on ideas developed from our analysis of the traditional ramped half and half (RHH) technique [4]), which produces 100% effective code [5] at initialisation. We present not only the performance results generated when using this algorithm, but also the size and shape metrics of the programs produced by SDI and compare them to the RHH technique over a range of benchmark problems. We hybridise our SDI algorithm with the existing FULL [4] initialisation technique and present further results which, when compared to those of the RHH and SDI techniques, demonstrate that full semantic diversity is not the only major influence on program initialisation for GP.

Finally, in order to examine our theory that the shape of the program tree at initialisation will influence performance, we conduct an experiment designed to test whether different shapes of trees with the same semantics will influence the performance of GP using our semantic models.

Section 2 reviews related literature in the field. Section 3 presents the algorithms we use for program initialisation. In Sect. 4 we present our results, which include unique behaviour analysis, program bias analysis, program metrics analysis, GP performance results and evolvable shape experiments over a range of benchmark problems. In Sect. 5 we present a discussion of the results and in Sect. 6 we present conclusions. In Sect. 7 we discuss several avenues for future related work.

2 Review of related literature

In this section we review a number of issues concerned with program initialisation in the context of this work. We review existing issues and techniques within program initialisation, the concept of program diversity and issues relating to program structure (or tree shape).

2.1 Existing initialisation techniques

The standard method for population initialisation, which we analyse in this paper, is the RHH method. This was introduced in 1992 by Koza [4], who set out three methods for creating a diverse population: GROW, FULL and RHH. Koza chose to

use the RHH technique for the majority of his experiments after conducting several experiments comparing FULL and GROW to the RHH because "...the ramped half-and-half method creates trees having a wide variety of sizes and shapes." [4, p. 93]. Koza also recognized that whilst the RHH method created a variety of programs, there was the possibility that programs could be duplicated. Therefore, he added syntactic duplicate checking of the programs to ensure the syntactic diversity of the starting population.

A *language bias* (defined by Whigham as "bias is the set of all factors that influence the form of each program" [6]) is present when there is a bias in the choice of items from the function or terminal sets. In 1995, Whigham [6–8] analysed such a bias and its effects on grammatically based genetic programming. Whigham conducted experiments that explicitly added segment(s) of code to a program in the population. For example, in one experiment he biased *if* statements such that the condition could only be a specific terminal. This narrowed the exploration of the search space and increased the probability of finding an ideal solution by artificially including segments of a known perfect solution. This demonstrated that specific language bias could be beneficial but could also severely compromise the ability of GP to find a solution to a problem with the highest fitness.

In 1996, Iba [9] devised an approximately uniform tree generation method (RAND_TREE) to combat the idea of language bias. In parallel to Iba's efforts, Bohm and Geyer-Schulz [10] independently devised a method called *Exact Uniform initialisation* based on statistical theory with the same objective in mind. Both of these papers report slight improvements in the success of GP runs compared to RHH. In these papers, however, only a small number of examples were studied, which left their results open to issues of problem-dependence (as noted by Bohm and Geyer-Schultz). This issue was addressed in 2001 when Luke and Panait [11] conducted a more comprehensive survey and comparison of population initialisation methods. This survey concluded that there was no significant statistical difference in performance between the RHH and the uniform initialisation methods.

In 2000, Langdon [12] developed ramped uniform initialisation as part of experiments to control code bloat through changing the bias in the distribution of syntax. The algorithm is similar to Iba's and focuses upon a method of distributing syntax, rather than controlling the semantics of programs. Another potential approach for resolving this type of question would be to ask whether the RHH and uniform creation methods create a similar behavioural bias. In addition, this approach could additionally explain why a theoretically grounded uniform initialisation method cannot improve results when compared to an ad hoc method such as RHH.

Despite the risk of imposing bias in a starting population, it is still a requirement that the GP practitioner has some control over the size of the programs produced (measured in depth or length) in order to prevent excessively large programs being produced. Producing a starting population should take a relatively short amount of time allowing for the fact that GP runs need to be executed many times to provide some degree of consistency in the results. To address this issue, Chellapilla [13] devised an algorithm called RANDOMBRANCH which utilized a specified length rather than depth and produced approximately uniform programs. One problem with this algorithm (highlighted by Luke and Panait [11]) is that because the RANDOMBRANCH

algorithm divides up the branch depths evenly, there are many trees that this initialisation method cannot produce. This would result in a language bias in the starting population; however, the effects on behavioural bias remain unstudied.

A later effort by Luke in 2000 [14] addressed the related issue of control over program initialisation. This resulted in two *Probabilistic Tree Creation* algorithms known as PTC1 and PTC2. These algorithms differed from those previously mentioned in that they allowed more user control. PTC1 allowed the user to provide the probability of appearance of individual functions as well as to define an average size of the initial programs. However, this method does not give the user any control of the variance of these programs. PTC2 addresses this issue by allowing the user to set a probability distribution of tree sizes which gives control over the variance in tree sizes. In comparison to the uniform based algorithms, PTC1 and PTC2 are simpler to implement and provide the user with much more control over the size and variation of programs in the initial population. In a similar way to Whigham's work, PTC1 and PTC2 give the user the kind of bias that could focus the starting population more towards where a global optimum will occur. The danger with this is that if the wrong kind of bias is used, then the exploration could be steered away from the global optimum.

In 2007, Looks [15] proposed a heuristic to increase behavioural diversity during program initialisation, which was demonstrated to outperform the RHH technique in terms of both success rate and overall computational effort. This shows that behavioural diversity does have an important role to play in population creation. In this paper we present further analysis, using a different technique to that of Looks [15], in which we not only enumerate unique and duplicated programs, but we also examine in detail the types of behaviours frequently produced by the RHH initialisation method. In addition, we conduct a comparison of the size and shape of the programs generated by the SDI, HSDI and RHH techniques.

2.2 Program diversity

When considering diversity in starting populations, it is important to understand two distinct types of diversity. The first type is syntactic diversity, that is, programs in the population being syntactically different. Koza [4] argues that this is important both as a method of generating programs with different behaviours, and as a means of providing a pool of material from which programs can be evolved. The second type is behavioural diversity, that is, diversity of the input–output behaviour. It is easy to find examples of sets of programs that are all syntactically distinct, yet which have identical behaviours.

Whilst Looks [15] introduced semantic diversity to program initialisation, studies of semantic diversity are not new to GP. Gustafson et al. [16–18] conducted multiple analyses of behavioural diversity in GP. In 2004, Gustafson et al. [17] conducted an analysis comparing behavioural diversity measures with fitness. These behavioural measures were based on two edit distances and this analysis concluded that edit distance showed a strong correlation with fitness difference. Gustafson et al. [16] present three different methods for measuring the behaviours of the programs they study. However, the authors mention that even these mechanisms do not provide an exhaustive presentation of the behaviour of the programs. One of the

limitations of Gustafson's work [16, 17] was that a behaviourally canonical representation was not used to check for isomorphism; the use of ROBDDs (Reduced Ordered Binary Decision Diagrams) and the ant behaviour reduction algorithm in our work gives us that very ability.

In earlier works, Poli and Langdon [19] and O'Reilly and Oppacher [20] discuss program diversity through analysing how different crossover and search operators perform on different problems, with varying results. In this paper, we analyse these issues using novel semantic analysis methods.

2.3 Program structure

Previous work on the analysis of changes to program structure has demonstrated that tree shape does have an influence on the success or failure of GP. Punch et al. [1] and Gustafson et al. [3] present artificial problem domains in the form of royal trees and the tree-string problem. These artificial problem domains are tunable and designed to evaluate the relationship between program structure and the ability of GP search to navigate through the search space.

Whilst Langdon et al. [21] provide evidence that programs evolve towards particular shapes, Daida et al. [2, 22] suggest not only that program structure has a role to play in evolution, but that predictions can be made as to which shapes are the most evolvable. Our analysis (in Sect. 4) tests two extremes of different program shapes at the point of program initialisation. By introducing this kind of structure analysis at initialisation, we can separate the effects of shape at initialisation from those effects which are due to evolution.

A further well documented aspect of program structure is the intron phenomenon [5, 23, 24]. Introns form either unreachable or redundant code within program trees. There is some disagreement as to the link with program bloat: some authors [25, 26] suggest a positive link between introns and code bloat, whilst others [23] disagree. In a structural sense, authors have suggested that introns are required [27] in order to protect areas of valuable fitness within a program tree. Additionally, authors have provided evidence that fitness neutral evolution is valuable to GP [28]. In relation to this work, we present algorithms capable of producing starting populations which are intron free, and we examine these effects in our discussion.

3 Methods and algorithms

3.1 A general framework

The aim of our work is to demonstrate the effect of moving the initialisation of programs in GP away from random combinations of syntax and towards semantic building blocks of code. This goal is not as easy to implement compared to randomly combining syntax. Furthermore, the semantic initialisation process needs to be comparable in execution speed terms to the traditional ramped half and half initialisation.

The major issues within SDI are increasing semantic variety and producing fully effective starting programs. We present a behavioural analysis in Sect. 4; however,

we first set out the abstraction techniques and algorithms we use to enable semantically driven initialisation.

In order to control semantics we need to have a representation of the behaviour of programs. These behavioural representations are dependent on specific problem domains. In our experiments we use six different benchmark problems from the Boolean domain and one from the artificial ant domain.

The first step in the framework is to seed behaviours in the abstract representation. Once this has taken place, we then combine the behavioural representation at the root rather than changing other areas of the representation. This gives us the ability to build more complex behaviours quickly. New behaviours are added to the population if and only if they are unique. Once we have attained the new behaviours, we translate them into syntax as specified for the GP problem.

3.2 The problem domains

In our experiments we use seven test problems, which are described in this section.

The objective of the 6 bit multiplexer problem is to interpret two control bits {A0, A1} as a binary number and choose the correct output bit {D0, D1, D2, D3} based on the number. The fitness is the number of correct choices over all possible 64 combinations of inputs for the 6 Boolean bits. The function set is {IF, AND, OR, NOT} and the terminal set is {A0, A1, D0, D1, D2, D3}. The 11 bit multiplexer is similar to the 6 bit multiplexer. However, there are three control bits which, represented as a binary number, can select one of 8 outputs. The 11 bit multiplexer is substantially more complex compared to the 6 bit multiplexer as the size of the search space increases from 2^{2^6} (6 bit multiplexer) to $2^{2^{11}}$ (11 bit multiplexer). The function set of the 11 bit multiplexer is the same as the 6 bit multiplexer and the terminals are {A0, A1, A2, D0, D1, D2, D3, D4, D5, D6, D7}.

The objective of the even 4 parity problem is to return true if and only if an even number of the inputs are true. The function set is the same as for the multiplexers and the terminal set is {D0, D1, D2, D3}. The 7 parity problem is an extension of the 4 parity problem with the same objective and function set. The terminal set is {D0, D1, D2, D3, D4, D5, D6}.

The objective of the 5 majority problem is to return true if and only if the majority of the inputs are true. The function set is the same as the multiplexers and the terminal set is {D0, D1, D2, D3, D4}. The 9 majority problem is an extension of the 5 majority experiment with the same function set and the terminal set {D0, D1, D2, D3, D4, D5, D6, D7, D8}.

The artificial ant domain models an ant operating over a trail of food pellets on a grid. The ant must collect all the food pellets in order to achieve full score. We use the benchmark *santa fe* trail [29] which represents 89 food pellets in a broken trail on a 32 × 32 toroidal grid. The function set for the ant problem is {IF-FOOD-AHEAD, PROGN2, PROGN3} and the terminal set is {MOVE, TURN-LEFT, TURN-RIGHT}. The function IF-FOOD-AHEAD is an if-then-else structure with the condition representing whether the ant has a food pellet in the grid square directly in front of it. PROGN2 and PROGN3 execute the instructions they hold in

sequence. The only difference between them is that PROGN2 has an arity of two and PROGN3 has an arity of three.

3.3 Boolean domains

To enable us to analyse semantic characteristics of Boolean programs we use a Java implementation of GP [30], linked to the *Colorado University Decision Diagram Package* (CUDD—[31]) using the *JavaBDD* [32] interface.

The important functionality that this provides is the ability to reduce program representation by removing redundant and unreachable arguments. We can obtain canonical representations known as ROBDDs [33] of the behaviour of the Boolean programs: this allows us to compare programs for semantic equivalence. Any two programs that reduce to the same ROBDD are semantically equivalent, and *vice versa*.

An ROBDD is a node tree where each node represents a Boolean decision variable. These nodes are linked by true and false branches to either other nodes or the final output of the diagram (true or false). Bryant [33] describes a method (and algorithm) to reduce the binary trees to a canonical form. An example of an ROBDD can be found in Fig. 1.

Two important measurements we use in the analysis of ROBDDs are *SatCount* and *NodeCount*. *SatCount* is a value between 0 and 1 that represents the number of input combinations resolving to true in the ROBDD, divided by the total number of input combinations possible. *NodeCount* will return the number of variables present in the ROBDD (in the GP context the number of terminals). This function can be used in conjunction with *SatCount* to classify behaviours. For example, a *SatCount* of 0.25 with a *NodeCount* of 2 would represent a function such as AND A0 A1, given that there are four input combinations in total of which only one results in

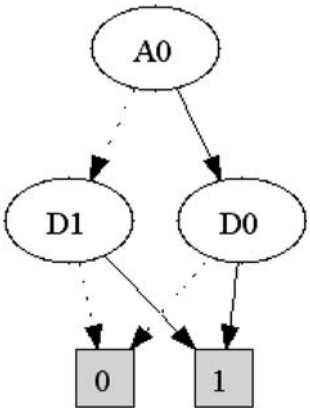

Fig. 1 This example ROBDD is a canonical representation of behaviour. In the diagram, *circles* represent variables (terminals in the GP context); *solid arrows* represent true paths; *dotted arrows* represent false paths. The *squares* marked 1 and 0 represent output of true and false respectively. This behaviour could be represented by many different parse trees. Two examples of parse trees that would result in this behaviour are IF A0 D0 D1 and IF (NOT A0) D1 D0

true. In a second example, a *SatCount* of 0.75 with a *NodeCount* of 2 would indicate a function such as OR A0 A1.

A *tautology* is a program which produces the output *true* regardless of input. In the case of a tautology, the value of *SatCount* is 1. A *contradiction* is a program which produces the output *false* regardless of input. For a contradiction, the value of *SatCount* is 0. Unlike in program parse trees, ROBDD functions are not represented explicitly as nodes in the parse tree, but by using true and false links between the variables. Due to the reduction mechanism [33], it is possible to reduce some ROBDDs to just true or false (i.e. tautology or contradiction). If one considers the example AND A1 (NOT A1), this program will always result in false and the ROBDD of this program will reduce to false. In the GP context this is very undesirable because it indicates that the result is not dependent on any of the variables and always returns the same answer (true or false).

If we consider the tautology and contradiction, the *NodeCount* will be 0. If the *NodeCount* is 1 and the *SatCount* is 0.5 we know the ROBDD of the program parse tree will reduce to just one variable (terminal) and possibly a function such as NOT.

3.3.1 SDI for Boolean domains

The SDI algorithm is designed to create a population consisting of semantically distinct programs, by using the ROBDD method described above as a way of checking whether newly generated programs are semantically equivalent to programs that have already been placed in the starting population. Pseudo code for the algorithm is presented below:

```
Phase 1:
for each Terminal in List Of Terminals
   create ROBDD representation of Terminal
   add to ROBDD_Store
end for each
Phase 2:
while ROBDD_Store size < population size
   choose a random function from the function set
   choose 1, 2 or 3 (dependent on function) ROBDDs from the ROBDD_Store
   at random (uniform probability)
   apply the function to the ROBDDs at tree root
   if resulting ROBDD is a new behaviour and not a tautology or contradiction
      add resulting ROBDD to ROBDD_Store
   end if
end while
Phase 3:
for each ROBDD in ROBDD_Store
   translate ROBDD to program
   save program
end for each
```

In phase 1 we represent the terminals as ROBDDs and add them to the ROBDD_Store. This is required as we need at least one example of each input variable to be present as building blocks for phase 2. Without any ROBDDs in the ROBDD_Store from phase 1, phase 2 would fail as it would have nothing to combine using the functions. Phase 2 starts to combine the terminals (or single variable ROBDDs) using functions: when a semantically unique function is produced, it is saved in the ROBDD_Store. As phase 2 continues, the algorithm is able to take advantage of all of the representations held in the ROBDD_Store and this encourages more complex behaviour to be generated as the algorithm continues. Phase 3 translates the ROBDDs back to Boolean parse trees. One other important factor is that this algorithm will not produce tautologies or contradictions.

The behavioural model works in the scenarios we present; however, it is not without limitations. Bryant [33] noted that large graphs (in excess of 100,000 vertices) were possible once 10 variables had been exceeded (in the GP context, terminals). This limitation is manageable with the combination of Bryant's reduction mechanism and increased power in modern PCs. As such, the generation of a starting population for the 11 bit multiplexer took a matter of seconds. It should be noted that the barriers (with more than 11 terminals) we faced were not due to the creation of ROBDDs, but to the translation mechanism which transforms the behaviours into syntax.

Close observers of the SDI algorithm will notice that there is no syntax that will result in the guarantee of termination. Termination in this case occurs because the behavioural search space is larger than the number of programs being initialised. In GP, it would be very unlikely that a practitioner would be trying to evolve a solution when they could generate every possible behaviour in a reasonable amount of time, otherwise it would defeat the point of using GP to solve a particular problem. Theoretically, for small problems (such as a 3 bit multiplexer), if the SDI was asked to generate a population of more than 254 behaviours (256 in total minus the tautology and contradiction), the algorithm would not terminate.

3.4 Ant domain

In order to represent the behaviour of ants we consider a behavioural model as a sequence of moves and orientations that represent the path which the ant has travelled during only one execution of the ant control program (or GP candidate solution). When the artificial ant is simulated in GP, we execute the candidate solution until the ant has travelled a set number of time steps (600 in this case) [29]; although, in this model we are only interested in a single execution of the ant control code. In addition to this, we execute the ant code on a toroidal grid (32×32) that contains no food pellets and we calculate the path of both the true and false branches of the IF-FOOD-AHEAD (if-then-else) function.

An example program in this domain is as follows:

PROGN2 (PROGN3 (MOVE, (IF-FOOD-AHEAD (PROGN2 (MOVE, TURN-RIGHT)) MOVE) MOVE) TURN-LEFT)

An example of the syntax, equivalent to the above program, we use is as follows:

$$\text{Ant representation} = \langle \mathrm{M}, \langle \mathrm{M}, \mathrm{S} \rangle, \langle \mathrm{M} \rangle, \mathrm{M}, \mathrm{N} \rangle$$

The character M represents one move and the characters N, S, E, W represent the orientations north, south, east and west respectively. The subsequences within the set indicate when a branch of an IF-FOOD-AHEAD statement is being accessed and coordinates within those brackets indicate the path travelled during each branch of the condition. Because we are only concerned with modelling the shape of the trail (consider the picture of the trail as we look down on the grid), it is unimportant whether or not the ends of the if-blocks have different orientations for the trail to continue upon. Therefore, at the end of the conditions we reset the current orientation to the orientation before the ant entered the if branches. The reason for this reset is that we are only interested in the relative meaning for modelling change of position (or picture of the trail).

More formally, we can describe in Backus–Naur Format the structure of a representation:

$$rep ::= \langle < expr > \rangle \tag{1}$$

$$expr ::= M|N|S|E|W| < bracketExpr > | < expr > , < expr > \tag{2}$$

$$bracketExpr ::= \langle < expr > , < expr > \rangle \tag{3}$$

In addition to this model structure, we add three checks which condense the abstract representation. The first check is to remove duplicate sub-branches of the same *if* statement and incorporate the paths as part the fixed path the ant was on before the *if* statement. The second check searches for sequences of orientations and reduces them to the last orientation in the sequence. This has the effect of removing redundant turns from the ant abstract model. The final check, moves through the representation, remembers the current orientation and removes any duplicate calls to turn to the current orientation. This serves to remove redundant turn instructions.

There are underlying differences between the Boolean domain, which is finite, and the ant domain, which is toroidal, and therefore potentially infinite. When a domain is infinite, it is necessary to constrain the size. We have done this by constraining the behaviour, whereas in previous work [4] this has been done by constraining the syntax. Furthermore, whilst programs in the Boolean domain would be run only once, in the ant domain the program is executed repeatedly up to a limit of 600 time steps. As a result of both the toroidal nature and the repeated executions, a behavioural size limit of 10 moves (chosen as an arbitrary reasonable value) has been applied to enforce a syntactic size limit. This limit was set so that if the function PROGN3 is used, it allows enough moves to traverse the grid in one execution.

3.4.1 SDI for ant domain

Pseudo code for the algorithm is presented below.

```
Phase 1:
Generate the 4 core behaviours (see discussion below)
Add core behaviours to Behaviour_Store
Phase 2:
while Behaviour_Store size < population_size
  choose a random function from the function set
  choose 2 or 3 (dependent on function) Behaviours from the Behaviour_Store at random (uniform probability)[a]
  apply the function to the Behaviour_Store
  if resulting Behaviour is a new behaviour and has at least one move
    add resulting Behaviour to Behaviour_Store
  end if
end while
Phase 3:
for each Behaviour in Behaviour_Store
  translate Behaviour to program
  save program
end for each
```

[a] The algorithm will only choose programs with 10 moves or less

The core behaviours of phase one make up the very basic operations the ant can perform. As with the Boolean domain, phase 1 is needed to seed phase 2. These are a representation of one move in every direction. This acts to provide the ant with at least one instance of all the possible moves. Phase two builds up more complex programs from the building blocks and as more behaviours are generated and saved they can be used by the algorithm to build yet more complex behaviour. Phase three translates the behaviours back to syntactic representation for the GP. This algorithm will not produce behaviours that contain no moves.

As with the Boolean domain, there is no syntax to enforce termination. In this case termination is dependent again of the size of the behavioural search space being greater than the starting population.

3.5 Hybridised SDI

In addition to the SDI algorithm, we developed a hybridised version of the algorithm (HSDI). At one end of the scale we have the existing RHH algorithm which we will show produces repeated simplistic behaviours. At the other end of the scale we have the SDI which can produce complex behaviours with no regard for program size. A hybridised version of the algorithm would combine behaviours both in the simplistic and complex areas of the search space, aiding a wider search.

In the hybridised version of the SDI we alter phase 1 of the initialisation algorithm. Instead of using terminals or core behaviours as the initial seed to build on, we use the existing FULL [4] algorithm to create the first round of behaviours. We use FULL because of the increased semantic diversity it provides (shown in Fig. 3) and the fact that because we are converting the programs into behaviours we are not concerned by the shape of the trees produced at this stage. Upon creation of each FULL program we store the behaviour, if and only if, it is unique and is not a tautology or contradiction (or contains no moves). Section 4.2 shows results to demonstrate the current level of unique behaviours within different starting populations using the RHH and FULL techniques. Once this seed is complete, the HSDI algorithm continues in the same way as the SDI algorithm and combines behaviours at the root, therefore encouraging more complex behaviour. In phase 3, the behaviours are translated back to 100% effective syntax.

In the hybridised version of the artificial ant problem, phase 3 can become problematic due to the fact that the abstract representation can reduce branches of the IF-FOOD-AHEAD statement to nothing (for example if it contained a turn left and then a turn right instruction). As such, we made an addition to the ant syntax that can only occur in back translation which is a SKIP operation. This has no effect on the ant apart from costing it one move. The move cost is required because some IF structures result in the ant always falling into the SKIP function when it is executed.

3.6 Evolvable shape analysis algorithm

In order to evaluate how the shape of programs in starting populations affects evolution during GP runs we present a separate set of experiments. In these experiments, we compare how starting populations made up of FULL node trees with all branches reaching the same maximum depth perform against deeper thinner node trees created from our behavioural representations.

In order to analyse the effects of different tree shapes at the point of program initialisation, we developed two experimental algorithms. These algorithms are intended to be used as analysis tools only, and are not designed to be for general use. The first of these is a modified version of the FULL algorithm, in which tautologies are prevented, and the maximum depth is 4. This is to allow us to analyse the performance of fully-branched ("fat") trees. The algorithm for this modified version of FULL (*MODFULL*) is as follows:

```
while first_Gen < pop_Size {
  generate FULL_Program
  generate FULL_Program_Representation
  if FULL_Program_Representation is NOT a tautology or contradiction
    {
      add FULL_Program to first_Gen
    }
}
```

The second algorithm is designed to create trees in which programs are represented using minimalistic ("thin") representations. This is achieved by creating programs using the MODFULL algorithm and then applying a "washing" process to them that removes redundant and unreachable code. The washing process works by translating the code into a canonical representation, then translating that back to the tree representation. Pseudo-code for this washing step is as follows; the complete WASHED algorithm consists of taking the outputs from the above code fragment and then applying this process.

```
for each program in first_Gen {
  translate program to representation
  back-translate representation to reduced_program
  replace program with reduced_program
}
```

4 Results

In this section we present results for seven sets of experiments, all of which analyse some characteristic of the GP run by comparing different initialisation methods. These experiments are a speed comparison, analysis of unique behaviours, bias analysis, size and shape analysis, measure of overall GP performance, and analysis of evolvable shape.

4.1 Speed comparison of initialisation methods

To address initial fears that these new algorithms might take impractical amounts of computation time, we present a comparison of speed of initialisation showing the time it takes to initialise a starting population using SDI, HSDI and the traditional RHH technique.

Whilst Table 1 shows that the RHH technique takes less time to generate programs, this experiment does not give any indication of the comparable quality of the resulting programs. If we consider that the results quoted are in milliseconds, then it is reasonable to use semantically driven initialisation in order to attain a semantically diverse starting population, as even the slowest initialisation is only 6 s.

A second aspect of this experiment which is not comparable is that RHH has a built in depth limit and as such cannot build programs greater than that depth. As a result of these depth limitations we know that the average depth of programs will fall in the range of two to six, therefore limiting the size of the programs generated. The SDI algorithm does not use a limit[1] on the size of the programs. Therefore, it seems reasonable to expect that the process might take longer. It only represents the behaviour in the form of effective code. Arguably, the SDI is having to do more

[1] Except as already explained for the artificial ant SDI with the behavioural building block size of 10 moves.

Table 1 We initialise 100 populations of size 500 using SDI, HSDI and the RHH techniques

The times quoted are in milliseconds and represent the average number of milliseconds taken for one initialisation of a population to take place

Experiment	SDI	HSDI	RHH
Mux-6	414	415	87
Mux-11	5581	3882	86
Even-4	161	203	87
Even-7	650	614	90
Majority-5	243	279	89
Majority-9	1885	1516	87
AASF	548	625	129

work in terms of code generation, and the size and shape results in Sect. 4.4 support this. Overall, we do not consider the difference in speed of generation to be a significant factor in choosing between these different methods.

4.2 Behaviour in starting populations

4.2.1 Analysis of unique behaviours

We use the behavioural representations to analyse GP starting populations. Given a starting population, we convert each member of the population into behavioural model form. We enumerate the number of unique behaviours in the population by testing for model equivalence. In addition to this, we calculate the number of programs associated with a specific behaviour to analyse any bias towards specific behaviours. In these experiments we initialise 100 populations at each population size and all results reported are averages of these 100 initialisations.

4.2.2 Unique behaviours

Figure 2 shows that in every model, there is a notable percentage of duplicated behaviours. The even 4 parity model represents the worst performing result in terms of the number of unique behaviours: when the population increases past 2500, less than 40% of the programs produced by RHH are unique. Furthermore, the 11 bit multiplexer demonstrated the least duplication of behaviours with a maximum of 15%. One feature applicable to all models is that, at different levels, all percentages of unique programs decrease as the population increases. This could indicate a type of bias such that some behaviours are favoured and repeatedly produced by the RHH. The final feature of note is that as the number of terminals increases for the Boolean domain, there is less duplication of behaviour.

Figure 3 shows a sample of two of our test problems comparing the semantic diversity when using the FULL (depth 4) and RHH techniques. In both cases the FULL technique generated more semantic diversity and paired *T*-tests revealed that this difference is significant at the 95% and 99% confidence intervals (*P*-value of 0.000 for majority 5 and *P*-value of 0.001 for the 6 bit multiplexer). This is an interesting result as previous authors [4] have reported that FULL does not perform as well as RHH in terms of GP performance. This result lends weight to the theory

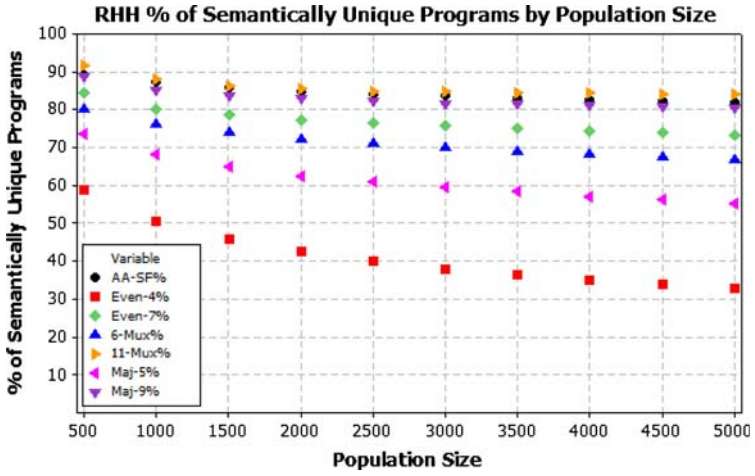

Fig. 2 Enumeration of unique behaviours present in starting populations expressed as a percentage of the total population size. This analysis is repeated for population sizes ranging from 500 to 5000 for each experiment. All results quoted are an average of 100 initialisations

that there is an ideal evolvable shape of program tree. In practical terms, if we require a semantically diverse seeding mechanism, then the shape of the programs is not important because they will be modelled as behaviour; therefore, this makes an ideal method to seed the HSDI.

4.3 Bias analysis

In order to examine bias more precisely, we run a second experiment which initialises 100 populations of size 1000. We record every behaviour produced by the 100,000 initialised programs and if behaviours are produced multiple times, we keep a record of the frequency.

Table 2 shows that for initialisation of population in the even 4 parity model, the RHH favours simplistic behaviours with the readings at rank one and two being the contradiction and tautology. As explained in Sect. 3.3, tautologies and contradictions are a special case as their result does not depend on the input value of any of the variables (or terminals in the GP context). In every initialisation of population size 1000, an average of 40.94 tautologies and 42.25 contradictions occur. This results in 8.32% (combination of tautology and contradiction) of the programs generated in each initialisation not depending on any of the input variables.

Moreover, none of the behaviours in the 10 most frequent behaviours contain all the terminals and therefore they result in partially blind candidate programs. Behaviours in ranks three to six represent single terminals with or without the possibility of a NOT function and the behaviours in ranks seven to 10 represent simple AND or OR functionality. The 46th ranked most frequent record (with a frequency of 2.92) is the first record which has a node count of four. This indicates that behaviours that use four inputs (and possibly all the terminals) are being infrequently created when compared to the simplistic structures we see in Table 2.

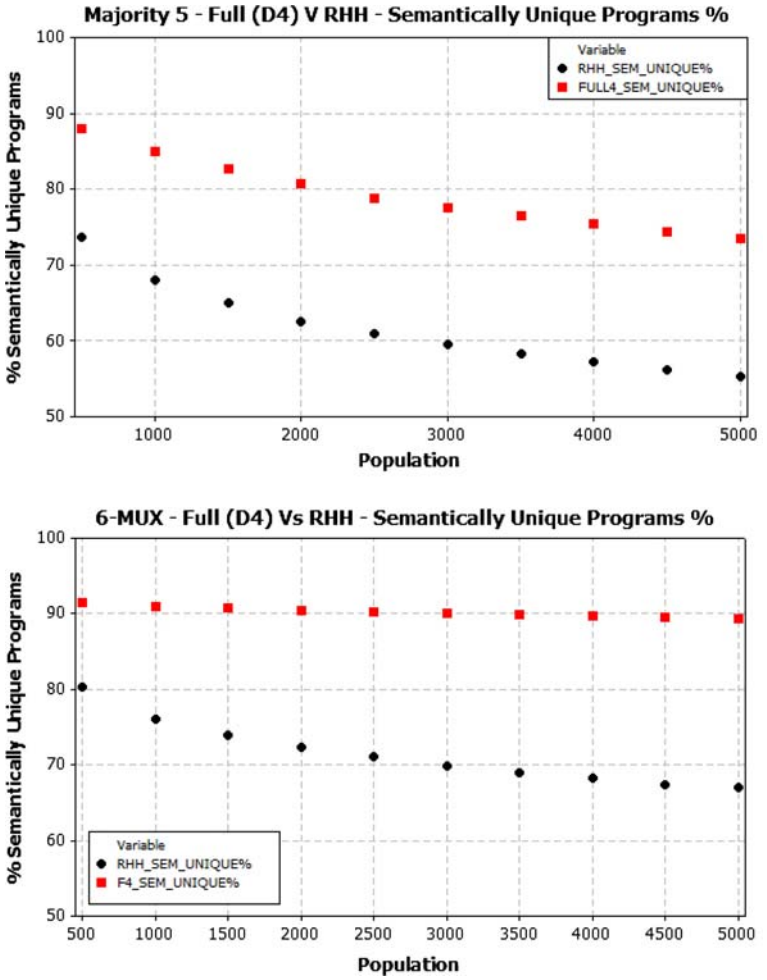

Fig. 3 The two graphs show the percentage of semantically unique programs generated by the FULL (depth 4) and RHH techniques against population size for the majority 5 and 6 bit multiplexer problems

In Table 2, the even 7 parity results are similar to that of the even 4 parity model. We see the tautology and contradiction states featuring at ranks one and two, and single terminals at positions three to nine and simplistic OR functionality in position ten. Whilst the behavioural structures are similar to the even 4 parity model, the frequencies are slightly reduced such that tautologies only represent 5.78% of the behaviours generated in an average initialisation. The introduction of more terminals adds but a little more diversity to creation of behaviours using the RHH technique. It is not until the 1515th ranked most frequent behaviour with a frequency of 0.5 occurrences per initialisation, that we see a node count of seven for the first time. Again, this indicates a bias towards simplistic behaviours being generated by the RHH technique.

Table 2 Bias results for the parity and multiplexer models

Rank	Frequency	Node count	Sat count
Even 4 parity			
1	42.25	0	0
2	40.94	0	1
3	18.59	1	0.5
4	18.32	1	0.5
5	18.14	1	0.5
6	18	1	0.5
7	10.7	2	0.25
8	10.55	2	0.75
9	10.53	2	0.75
10	10.5	2	0.25
Even 7 parity			
1	17.52	0	1
2	17.05	0	0
3	7.21	1	0.5
4	7.05	1	0.5
5	7.02	1	0.5
6	6.69	1	0.5
7	6.67	1	0.5
8	6.66	1	0.5
9	6.47	1	0.5
10	3.29	2	0.75
6 bit multiplexer			
1	21.78	0	1
2	21.68	0	0
3	9.15	1	0.5
4	8.93	1	0.5
5	8.85	1	0.5
6	8.7	1	0.5
7	8.65	1	0.5
8	8.64	1	0.5
9	4.13	2	0.25
10	4.1	2	0.25
11 bit multiplexer			
1	8.62	0	1
2	7.67	0	0
3	3.6	1	0.5
4	3.59	1	0.5
5	3.54	1	0.5
6	3.54	1	0.5
7	3.53	1	0.5
8	3.52	1	0.5

Table 2 continued

Rank	Frequency	Node count	Sat count
9	3.47	1	0.5
10	3.42	1	0.5

Rank indicates the relative frequency with 1 being the most frequent behaviour. The frequency is the total number of occurrences of this behaviour divided by 100 which indicates the number of times this behaviour is expected to occur per initialisation. Node count and sat count are as explained in Sect. 3.3

In Table 2, the 6 and 11 bit multiplexer experiments show similar results to that of the even parity experiments. Again, the tautology and contradiction states feature as the two most frequently constructed behaviours. These are followed by single terminals, and then simple two terminal structures. As the number of terminals in the problem increases, the chances of constructing a behaviour with all terminals present becomes even worse.

The first occurrence of a behaviour with a node count of six is ranked 559th, with a frequency of 0.13 occurrence per initialisation. This information is worth considering as the 6 bit multiplexer model frequently has its population size parameter set at 500. Therefore, if we consider an average initialisation, the population is unlikely to contain one candidate program which has all terminals present in the behaviour.

Table 2 shows that the bias results for the 11 bit multiplexer have similar characteristics to the other Boolean models: the increase in terminals results in a decrease in the frequency of behaviours which use all inputs. In our analysis, it was not until the 345th ranked most frequent program (with a frequency of 0.16) was reached until only three nodes were used to create a behaviour.

In keeping with the other Boolean domain experiments, Table 3 shows that the 5 and 9 majority experiments suffer a similar bias. In the case of the 5 majority experiment, it was the 150th most common behaviour with a frequency of 0.67 when a node count of 5 was first achieved. In the case of the 9 majority experiment, it was the 2614th most frequent behaviour before a node count of 9 was achieved.

The artificial ant results in Table 3 exhibit similar simplistic behaviours, albeit not in the same way as the problems in the Boolean domain. The most frequent structures assembled by the RHH technique are simple one move or turn structures. It is not until the 6th most frequent reading that a behaviour contains two operations. It is not crucial to have behaviours with large numbers of moves (because of re-execution of the ant control code), but in order to achieve full score, the ant will have to have a behaviour containing several moves and turns. To put this into perspective, it is not until the 197th most frequent reading (0.32 frequency) when five moves are first accomplished.

4.3.1 Discussion

Both the analysis of unique behaviours we present in starting populations, and the more in depth bias analysis for each model have revealed several biased features of the output of the RHH initialisation technique.

Table 3 Bias results for the artificial ant and majority models

Rank	Frequency	Move count	Final orientation
AASF			
1	11.11	1	E
2	9.82	0	S
3	9.70	0	N
4	7.97	0	W
5	7.93	0	E
6	5.48	2	E
7	5.19	1	S
8	5.14	1	N
9	5.13	1	N
10	4.96	1	S
Rank	Frequency	Node count	Sat count
5 Majority			
1	29.88	0	1
2	29.11	0	0
3	12.73	1	0.5
4	12.18	1	0.5
5	11.99	1	0.5
6	11.73	1	0.5
7	11.31	1	0.5
8	6.2	2	0.25
9	6.08	2	0.75
10	6.04	2	0.75
9 Majority			
1	11.26	0	0
2	10.61	0	1
3	5.01	1	0.5
4	4.88	1	0.5
5	4.84	1	0.5
6	4.76	1	0.5
7	4.6	1	0.5
8	4.57	1	0.5
9	4.46	1	0.5
10	4.42	1	0.5

Rank indicates the relative frequency with 1 being the most frequent behaviour. The frequency is the total number of occurrences of this behaviour divided by 100 which indicates the number of times this behaviour is expected to occur per initialisation. Node count and sat count are as explained in Sect. 3.3. The move count represents the number of moves in the behaviour and the final orientation is the direction the ant is facing on the grid after the moves have taken place

Despite preventing syntactically identical code being produced, there are still notable quantities of duplicated behaviours present in the starting population. Further bias analysis of all problem domains considered revealed that the RHH technique favours simplistic behaviours or tautology and contradiction states. Tautologies and contradictions in the Boolean domain represent the RHH's inability to create building blocks dependent on terminal values and its ability to produce ineffective code, as it is not possible to directly construct true or false with the syntax available. The fact that they are the most frequent behaviours in the Boolean problems indicates that this is the main weakness of using the RHH technique as an initialisation method.

If we consider the ant domain, a similar concept to the tautology or contradiction in the Boolean domain would be an ant that does not perform any moves. An ant could perform as many turns as it wanted without moving positions, but this would not allow it to achieve its ultimate goal of collecting food pellets. Unfortunately, this characteristic is the second to the fifth most produced behaviour by the RHH technique generating syntax in the artificial ant domain.

A final aspect of concern is the apparent inability of the RHH to construct more complex behaviour. In the case of the Boolean domain, we noted low rank of the first occurrences of candidate behaviours with node counts capable of representing all inputs present. With increasing numbers of terminals, it was harder for the RHH to generate behaviours that used all the inputs. This effect is not limited to the Boolean domain. If we consider the ant domain, it was not until the 197th most frequent behaviour (with frequency of 0.1) that the RHH achieved an ant that was capable of moving five positions.

An advantage of the SDI algorithm is that it knows when behaviours reduce to the tautology/contradiction/no move state, and therefore these behaviours can be removed. In addition the SDI will prevent bias in behaviours because it can enforce semantically uniqueness of programs in the population.

4.4 Size and shape analysis

4.4.1 Analysis of size and shape of programs

As previously mentioned there is no size or depth limit on programs produced using SDI. This section aims to present an analysis and comparison of the size and shapes of programs produced by both RHH and SDI. We initialised populations of size one thousand for each problem. We performed 100 initialisations and measured the average of the results. The metrics we used are program depth, program length (total number of nodes in the tree), number of functions, number of terminals and the number of distinct terminals.

4.4.2 Results

Table 4 shows size and shape analysis from the parity and multiplexer experiments. Paired *T*-tests revealed that all readings are significantly different at the 99% confidence level (*P*-value of 0.000). The two most notable points that apply to all

Table 4 Size and shape comparisons

	Depth	Length	Functions	Terminals	Distinct terminals	Length/Depth
Even-4						
SDI	3.8276	13.7575	6.7903	6.9672	3.8385	3.5943
RHH	3.6121	29.9810	14.4263	15.5546	3.2304	8.3002
HSDI	3.5902	12.003	5.9322	6.0710	3.6950	3.3433
Even-7						
SDI	7.3996	46.351	23.302	23.049	6.1663	6.2640
RHH	3.2604	28.197	13.537	14.661	4.3065	8.6483
HSDI	6.1480	32.348	16.230	16.118	5.5344	5.2615
6-Mux						
SDI	6.2213	31.8924	15.9970	15.8954	5.5051	5.1263
RHH	3.3571	28.5263	13.7093	14.8169	4.0049	8.4970
HSDI	5.2024	22.811	11.429	11.382	4.9242	4.3847
11-Mux						
SDI	14.2564	203.931	102.762	101.168	8.7484	14.3045
RHH	2.9887	27.199	13.041	14.158	5.2075	9.1006
HSDI	10.253	107.91	53.994	53.916	7.5249	10.5247

SDI readings produced by semantically driven initialisation, *RHH* readings produced by the ramped half and half technique, *HSDI* reading produced by the hybrid SDI. All readings quoted are averages of 100 runs of 1000 population size

but the 11 bit multiplexer are that the SDI produces deeper, thinner trees, compared to the RHH technique; and that the SDI increases the numbers of distinct terminals in all cases. The HSDI falls somewhere between the SDI and RHH extremes. In problems such as the 11 bit multiplexer this is useful as it creates smaller programs that are less likely to grow beyond the crossover depth cap [4] (in our case 17) during evolution as a result of the bloat phenomenon [27, 34, 35].

Table 5 shows the results for the majority and ant experiments. Paired *T*-tests revealed that all readings are significantly different at the 99% confidence level (*P*-value of 0.000). The majority experiments reflect the multiplexer and parity results in that the SDI produces deeper, thinner trees, except for the larger 9 majority problem and the 11 bit multiplexer. The 11 bit multiplexer and 9 majority results show a large increase in the size (all metrics) of the programs produced. This relates to the earlier discussion in Sect. 3.3 regarding the limitations of generating syntax directly from behaviour. As there are no size checks in the SDI, it will produce syntax to represent the behavioural complexity of the problem faced. In the majority experiments (Table 5) the SDI and HSDI consistently produce more distinct terminals during the initialisation which indicates a better ability for programs to deal with all possible inputs presented. Similar characteristics of size and shape are shown in the ant domain. Deeper thinner programs are produced by the SDI and HSDI compared to the RHH technique and both the SDI and HSDI produce a statistically higher level of distinct terminals.

Table 5 Size and shape comparisons

	Depth	Length	Functions	Terminals	Distinct terminals	Length/Depth
5-Majority						
SDI	4.9326	20.6616	10.3361	10.3255	4.6724	4.1888
RHH	3.4762	29.1390	14.0198	15.1193	3.6557	8.3824
HSDI	4.3329	16.141	8.0663	8.0745	4.3020	3.7252
9-Majority						
SDI	10.5130	99.9944	50.3765	49.6179	7.5707	9.5115
RHH	3.1005	27.4803	13.1808	14.2995	4.7949	8.8632
HSDI	8.1057	61.135	30.632	30.503	6.6202	7.5422
AASF						
SDI	6.3281	49.542	19.486	30.056	2.9631	7.8289
RHH	3.7340	56.369	23.718	32.652	2.7955	15.0961
HSDI	5.4073	37.788	15.392	21.867	2.9906	6.9883

SDI readings produced by semantically driven initialisation, *RHH* readings produced by the ramped half and half technique, *HSDI* reading produced by the hybrid SDI. All readings quoted are averages of 100 runs of 1000 population size

One final point to mention is the intron wash effect the HSDI will have on the populations it produces. It is seeded from the FULL method, but generated from behavioural representation of that code, so that it produces effective code, which, from studying the size and shape analysis, is supported by the consistently low length/depth values in Tables 4 and 5 (with the exception of the 11 bit multiplexer).

4.4.3 Discussion

One global feature of the size and shape experiments is that no matter that the problem domain, the RHH always produces roughly similar sized programs, whereas the SDI produces different sized programs depending on the problem. It is reasonable to assume that this happens because different problem domains have different behavioural requirements, therefore this would result in different size and shape characteristics of programs.

In addition to this, as the number of terminals increases, the potential combinations of behaviour will increase, causing deeper programs that require more functions and terminals to attain specific behaviours. This is borne out in our results for the Boolean domain, where the increase in program sizes and depths is in line with the increase in the numbers of terminals present in our experiments.

The second point to be made when studying these results is the level of importance that should be given to the measurement of the depth of the tree. All of the SDI and HSDI results produced trees of greater depth than the RHH (but generally shorter in terms of length) and this would be consistent with using composite functions to model specific behaviours. The length metric may be a better measure to use, in terms of flexibility, as it gives programs the opportunity to develop complex behaviour as well as controlling the overall size of the program.

The third observation is that in all experiments the SDI produced significantly more distinct terminals. Statistically, there are fewer possible behaviours that can be generated in the Boolean domain when not all the terminals are used. Once these are generated, the SDI has to generate more complex behaviour to retain the semantic variety it promises. As a result of this the SDI will automatically have a parsimony towards producing programs with all the terminals present, especially in larger populations.

4.5 GP performance results

In the experiments presented in this section, we compare the performance of GP runs that are initialised using the three methods, SDI, HSDI and RHH. We keep all parameters the same except for the population generation method.

We used the following GP parameters: 10% elitist reproduction; 100 runs of 50 generations; maximum depth 17; RHH depth 2–6; SDI or HSDI initialisation; crossover with 90% bias on functions and 10% on terminals; 0.9 crossover; 0 mutation (to remove additional variables from the experiment); and 7 competitor tournament selection. A population of 500 was used for the 6 bit multiplexer, even 4 parity, 5 majority and artificial ant (Santa Fe) experiments. A population of 4000 was used for the 11 bit multiplexer, even 7 parity and 9 majority problems.

Table 6 shows that the performance of the RHH, SDI and HSDI techniques vary depending on the problem being analysed. In this case, in overall terms we see the HSDI performing best in three experiments, the SDI in two and the RHH in two experiments (at the 95% confidence level). This raises two complex questions about how the dynamics of creating the starting population can impact on the performance of GP runs.

The first issue concerns the way in which we understand the distribution of initialised programs in relation to the search space for a particular problem. The SDI performs well on the parity and 6 bit multiplexer experiments, but poorly on the 11 bit multiplexer experiment, when compared to the RHH. A simplistic theory is that the SDI produces too many complex behaviours in the wrong region of the search space when we may be looking for simplistic programs in another region of the search space. This might explain why, for example, the SDI performs well at the 6 bit multiplexer, but poorly in the 5 majority experiment. It might also explain how the HSDI is able perform well on both multiplexers, as it is seeded with simplistic behaviour, but can build more complex behaviour on top of this.

The second issue is the effect of initialising 100% effective code compared to code with redundant and unreachable statements. We have already shown that the introduction of the SKIP operation was necessary in the HSDI applied to the artificial ant problem, to alleviate the problem whereby RHH and FULL produce dead branches for the IF-FOOD-AHEAD statement. Given the performance results shows in Table 6, this SKIP statement is clearly required.

The obvious comparison to this work is that of Looks [15] that presented results for a selection of multiplexer and parity experiments. Whilst Looks examines similar problems, he uses a different function set and different parameters, and a different semantic sampling initialisation. Despite these differences, for the 6 bit

Table 6 Performance of the RHH, SDI and HSDI techniques

Exp	Init	Max Scores	PT	Max Score G50	2T-50	Success
6-Mux	SDI	0.0287 ± 0.0534	–	0.0052 ± 0.0232	0.041	95% G4
	HSDI	0.0212 ± 0.0506	0.00	0.0003 ± 0.0031	0.041	99% G3
	RHH	0.0763 ± 0.0498	–	0.0431 ± 0.0529	–	51% G6
11-Mux	SDI	0.1411 ± 0.0790	–	0.0755 ± 0.0660	–	22% G25
	HSDI	0.0992 ± 0.0848	0.04	0.0377 ± 0.0441	S	45% G16
	RHH	0.1019 ± 0.0906	0.04	0.0339 ± 0.0401	S	45% G15
4-Par	SDI	0.0500 ± 0.0542	0.00	0.0200 ± 0.0396	S	77% G3
	HSDI	0.0601 ± 0.0590	–	0.0238 ± 0.0477	S	77% G6
	RHH	0.0949 ± 0.0735	–	0.0425 ± 0.0494	–	50% G9
7-Par	SDI	0.2966 ± 0.0822	0.00	0.1745 ± 0.0388	0.00	0% –
	HSDI	0.3176 ± 0.0775	–	0.2037 ± 0.0361	–	0% –
	RHH	0.3452 ± 0.0621	–	0.2582 ± 0.0297	–	0% –
Maj-5	SDI	0.0549 ± 0.0363	–	0.0300 ± 0.0266	–	32% G9
	HSDI	0.0467 ± 0.0356	–	0.0213 ± 0.0250	S	50% G9
	RHH	0.0427 ± 0.0394	0.00	0.0188 ± 0.0231	S	54% G10
Maj-9	SDI	0.1783 ± 0.0379	–	0.1338 ± 0.0120	–	0% –
	HSDI	0.1657 ± 0.0381	–	0.1202 ± 0.0124	–	0% –
	RHH	0.1228 ± 0.0467	0.00	0.0722 ± 0.0118	0.00	0% –
AASF	SDI	0.3220 ± 0.0547	–	0.2781 ± 0.0855	–	0% –
	HSDI	0.2889 ± 0.0639	0.00	0.2407 ± 0.1187	S	10% G7
	RHH	0.3177 ± 0.0812	–	0.2579 ± 0.1299	S	11% G5

Exp is the problem being analysed. Init shows the initialisation method. Max Scores shows the average of 100 runs of standardised maximum scores. The scores have been normalised to be in the range 0–1 for easy comparison. We use standardised fitness so 0 is the best value. The ± values are the standard deviation of the maximum scores normalised to the 0–1 scale. PT shows the result (*P* value) of a paired *T*-test comparing the SDI, HSDI and RHH results. A quoted *P* value of 0.00 is aligned against the best performing experiment and where the *P* value falls in the 95–99% confidence interval. The *P* value is quoted aligned with both experiments. If the scores are statistically the same, we mark it with S. Max Score G50 shows the maximum scores at generation 50 and 2T-50 shows the *P* value result of a two sample *T*-test of these values. Success shows the percentage of runs that reach full score and the earliest generation that full scores is reached out of all 100 runs

multiplexer and the 4 parity experiments, we still see a similar relative change in performance based on semantic style sampling. Looks uses a 5 parity and we use a 7 parity; given that our results support the value of semantic initialization for parity problems, our results are in line with his. The 11 bit multiplexer is different to the results of Looks; we can speculate that this is because our experiments use no depth limits on the SDI, and as a result (supported by the size and shape results (Table 4)) vastly increased program sizes are created (changing the distribution of programs in the search space). By contrast, Looks does control the size of the programs he produces using the semantic sampling algorithm. As stated in our introduction, our algorithms were designed to test theories in order to better understand the important issues in producing good quality starting populations.

Finally, there is an element of difference in program sizes when comparing the SDI, HSDI and RHH initialisation algorithms. Our size and shape results (Sect. 4.4) show that the SDI and HSDI algorithms produce programs at varying sizes depending on the problem. Some are smaller than the RHH output and some are larger. There is an argument to try to make the programs the same size for comparison, however, as the only control mechanism on the RHH algorithm is depth and we know that the HSDI and SDI tend to produce deep thin trees, it makes the job of making similar sized starting programs very complex. As such, we have made use of the traditional 2–6 depth range of the RHH algorithm.

4.6 Evolvable shape

In order to examine evolvable shape we use the MODFULL and WASHED algorithms, as set out in Sect. 3.6. The parameters are the same as in Sect. 4.5, however, we use the MODFULL and WASHED algorithms to examine the effect that changing the shape of programs without changing the semantics will have on GP performance. We use MODFULL and WASHED because we have seen in Tables 4 and 5 that the semantically reduced code will provide a dramatic contrast in tree shape, whilst retaining the same behaviour as the FULL code.

Table 7 demonstrates that changing the shape of the initialised program trees can have a dramatic effect on the performance of GP runs. If we consider the

Table 7 Effect of program shape on performance of GP runs

Exp	Init	Max Scores	PT	Max Score G50	2T-50	Success
6-Mux	WASHED	0.0235 ± 0.0506	0.00	0.0025 ± 0.0152	0.00	97% G4
	MODFULL	0.0663 ± 0.0577	–	0.0270 ± 0.0395	–	60% G8
11-Mux	WASHED	0.0971 ± 0.0850	0.00	0.0366 ± 0.0505	0.012	51% G18
	MODFULL	0.1498 ± 0.0879	–	0.0529 ± 0.0389	0.012	13% G33
4-Par	WASHED	0.0782 ± 0.0631	0.00	0.0381 ± 0.0554	S	60% G6
	MODFULL	0.0923 ± 0.0710	–	0.0444 ± 0.0513	S	50% G9
7-Par	WASHED	0.3177 ± 0.0784	0.00	0.2025 ± 0.0358	0.00	0% –
	MODFULL	0.3449 ± 0.0687	–	0.2491 ± 0.0527	–	0% –
5-Maj	WASHED	0.0503 ± 0.0346	–	0.0103 ± 0.0167	–	38% G9
	MODFULL	0.0321 ± 0.0373	0.00	0.0253 ± 0.0246	0.00	70% G7
9-Maj	WASHED	0.1656 ± 0.0383	–	0.1201 ± 0.0117	–	0% –
	MODFULL	0.1177 ± 0.0449	0.00	0.0683 ± 0.0083	0.00	0% –
AASF	WASHED	0.3032 ± 0.0625	0.011	0.2563 ± 0.1137	S	8% G2
	MODFULL	0.2968 ± 0.0789	0.011	0.2413 ± 0.1161	S	10% G7

Exp indicates the problem being analysed. Init represents the initialisation type (WASHED of MODFULL see Sect. 3.6). Max Scores is the average of the maximum scores ± the standard deviation of max scores. PT is a Paired *T*-test of the overall scores. 0.00 aligned with an experiment indicates a best result. In the event that results are statistically similar an S will feature and the actual *P* value will be quoted should results fall in 95–99% confidence interval range. Max Score G50 shows the average of the maximum scores at generation 50 and 2T-50 shows a 2 sample *T* test of these results. Success indicates the percentage of runs that reached full score at generation 50 and the first generation in which a single run attained full score

multiplexer, parity and majority experiments, all show a statistically significant difference between the MODFULL and WASHED algorithms overall, and all but one of the Boolean problems at the 95% confidence level at generation 50. However, whether MODFULL or WASHED gives a superior performance appears to be problem dependent.

The other factor of note when considering the success rates on the experiments that do find ideal solutions is the level of the difference in success. The biggest difference is 37% in the case of the 6-Mux. This shows the importance of program shape to evolvability in a GP run.

The artificial ant statistically favours MODFULL overall, and is statistically similar to WASHED at generation 50. This is not unsurprising, as despite the size and shape results in Table 5. the back translation mechanism still constructs full trees when it reassembles the ant movement instructions. The only difference is that redundant and unreachable code will have been removed in this reduced form. The fact the performance is statistically similar for the artificial ant indicates that simply removing both unreachable and redundant introns does not alter performance for this particular experiment.

5 Discussion

One of the main points to draw from this analysis, as well as the work of other authors (for example, [11, 15]) in the field, is that the choice of initialisation method may result in statistically significant variation in the performance of GP runs. In the context of this investigation, and the results we present in Tables 6 and 7, it is clear that changing different aspects of program initialisation can have an impact on performance far beyond that required for statistical significance.

5.1 Distribution of behaviours in the search space

Our early results in Sects. 4.2 and 4.3 clearly demonstrate how the existing RHH technique has bias, frequently duplicating simplistic behaviours, tautologies and contradictions (or no move ants). With these results in mind, we set out to counter these effects by producing the SDI algorithm in order to achieve complete behavioural diversity and build more complexity into the starting populations.

Preliminary analysis of the size and shape output (Tables 4 and 5) of the SDI algorithm appeared positive for two reasons. The first is the ability of the SDI to create starting populations that vary their size depending on the problem. This gives the impression that the SDI is actually modelling behaviours specific to the search space of each problem rather than the "one size fits all" solution in the RHH. The second is that depth appears not to be the best way to constrain programs. Our size and shape analysis (Tables 4, 5) showed that the SDI produced deeper but thinner trees in order to specifically model more complex behaviour.

Whilst the SDI might be theoretically superior in terms of distributing behaviours in the search space, Table 6 shows that it only outperforms the RHH technique in the six bit multiplexer and parity experiments. The parity problems are known to be

a deceptive problem [29], requiring a more intricate and complex program to solve them. This may be the reason that the SDI performed well on this problem. The multiplexer results are less clear, as it is intriguing that the SDI does not outperform RHH on the 11 bit multiplexer problem. A possible explanation for this is that in terms of the overall search space the program required to secure 100% fitness is relatively simplistic in comparison to all the behaviours in the 11 bit multiplexer search space. If one considers the exponential increase in search space size between the six bit multiplexer search space of 2^{2^6} to the 11 bit multiplexer search space of $2^{2^{11}}$, the SDI is having to model far more complexity to distribute programs through the breadth of the search space. It is possible that, as a result of this, the RHH is more biased to the correct area of the search space, therefore achieving a higher success rate.

The SDI failed on the majority experiments, which are arguably the most simple of the Boolean experiments. This would suggest that the RHH was better able to produce programs in the most successful search areas of the majority search space whereas the SDI's complexity worked against it.

Based on these results, the hybrid algorithm (HSDI) was created to take advantage of both the traditional FULL semantic generation and the increased complexity aspect of the SDI algorithm. The HSDI produced an algorithm capable of being the best performer overall in three experiments (at the 95% confidence level). In generation 50 it was able to statistically equal the best result in 4 out of 7 experiments, and was the best performing result in the 6 bit multiplexer at the 95% confidence level. The performance of the HSDI was disappointing considering it combined the different features of the SDI and RHH algorithms. This is a testament to the difficulty of creating a problem independent population generation algorithm.

5.2 Evolvable shape

Our results in Table 7 clearly show that the shape of the trees can have a significant effect on GP performance which is in agreement with Daida et al. [2, 22] and Langdon et al. [21]. In the Boolean domain, all of the experiments presented not only statistically significant differences (at the 95% confidence level), but also some dramatic changes in the performance of the GP runs. Given the variations in these results, it would appear that the ideal evolvable shape for a program is problem dependent and therefore it would be difficult to predict ideal program structures for specific problems. This suggests that further work comparing these results with those Daida and Hilss [22] would be valuable.

The similarity of the artificial ant results are unsurprising given that the translation mechanism between abstract meaning and syntax of the ant problem builds full trees (without redundant code) and, as a result, the experiment is comparing larger full trees with smaller, more effective, full trees. A noteworthy point is that the SDI contains no introns for the ant domain, whereas the HSDI has simulated introns (in the form of SKIP), and this increases the performance of the algorithm in the ant domain. A possible hypothesis is that being able to use the SKIP operator as one of the branches of the IF-FOOD-AHEAD operator statistically creates more opportunities for better performing solutions. This calls into question

how exactly we go about choosing the function set. In the case of the artificial ant, there may well be better choices of a function set.

Another algorithm that gives the user the ability to influence tree shape is Luke's *Probabilistic Tree Creation* [11, 14]. This is because the user-controlled bias in the appearance of functions and terminals would have an overall effect on tree shape. Luke's success using this algorithm could, at least partially, be a result of evolvable shape. With reference to the results we present in Table 7, it would be interesting to see if performance would increase if we change the probabilities of selection of the terminals and functions from experiment to experiment, for example, to cater for the difference between the parity and majority problems.

Having shown that there is a preferred evolvable shape for specific problems (Table 7), we can argue that this lends support to GP schema theory [29, 36, 37]. In our interpretation, the schemata are in a more abstract form, consisting of a tree of a (problem specific) particular shape containing "don't care" nodes. Alternatively, one could follow a strategy such as Salustowicz and Schmidhuber [38] enforcing a structure and using node weightings to perform the learning. This would be an interesting direction for future work.

The issue of problem dependence in the context of generating starting populations is a complex one. Table 6 shows that the three algorithms we present seem to favour particular problems and the results in Table 7 merely complicate the matter further. Based on this data, further research is required in the field of program initialisation in order to be in a position to recommend specific code generation algorithms for specific problems.

6 Conclusion

Our analysis of program initialisation in GP has shown that the initialisation method chosen can have a dramatic impact on the performance of GP runs. However, it appears that this impact is problem specific, and we cannot conclude that one of the algorithms that we have presented is best for every problem. We have shown some limitations of the RHH algorithm, and tested theories using our algorithms, but we have failed to find one clear solution to program initialisation that incorporates measures to deal with the behavioural distribution of programs and to control the shape of the syntax produced.

We present clear evidence that both the distribution of programs in the search space and the shape of the tree can have dramatic effects on the performance of GP. Both of these variables are strongly dependent on the problem being tackled, which therefore makes the challenge of constructing an initialisation algorithm a highly complex one.

As a result of this work, there is evidence to support the need to create initialisation algorithms that can explicitly exercise control over both the behavioural distribution of programs and the shape of the programs they produce.

Based on evidence we present, future work in the area of program initialisation needs to be able to address both behavioural diversity and program structures and the interactions between the two. A possible idea would be merging our algorithm

with Luke's PTC algorithms in order to gain a more granular control over program structure, whilst retaining behavioural diversity.

7 Future work

The first avenue to pursue in order to expand this work is to develop behavioural models for other domains, for example, symbolic regression. Further results using different problem domains may reveal different properties or create a clearer understanding of the relationship between semantic diversity and evolvable shape.

The second major area of research is to apply semantic control to other areas of GP. Semantic analyses of the crossover operation are starting to appear [39, 40]. A similar analysis could be used to study different mutation techniques, and also to compare the behavioural changes of mutation and crossover. This would provide data to help resolve the already contentious debate as to whether crossover is another type of mutation operator [41].

There is also the potential to create a type of semantic fitness function in order to measure the difference between two behaviours. This would have the potential of being very quick to execute in comparison to an input output fitness function when there is a large amount of input data used by the traditional fitness function.

Finally, with the use of semantic representation, we could construct a form of semantic pruning in order to attempt to cut bloat or test the performance of intron free GP compared to traditional GP.

Acknowledgement The authors would like to that the anonymous reviewers for their valuable feedback.

References

1. W.F. Punch, D. Zongker, E.D. Goodman, The royal tree problem, a benchmark for single and multiple population genetic programming, in *Advances in Genetic Programming 2*, Chap. 15, ed. by P.J. Angeline, K.E. Kinnear, Jr. (MIT Press, Cambridge, 1996), pp. 299–316
2. J.M. Daida, H. Li, R. Tang, A.M. Hilss, What makes a problem GP-hard? Validating a hypothesis of structural causes, in *Genetic and Evolutionary Computation—GECCO-2003*, Chicago, IL, ed. by E. Cantú-Paz, J.A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M.A. Potter, A.C. Schultz, K. Dowsland, N. Jonoska, J. Miller. Lecture Notes in Computer Science, vol. 2724 (Springer-Verlag, Berlin, 2003), pp. 1665–1677
3. S. Gustafson, E.K. Burke, N. Krasnogor, The tree-string problem: an artificial domain for structure and content search, in *Proceedings of the 8th European Conference on Genetic Programming*, Lausanne, Switzerland, 30 March–1 April 2005, ed. by M. Keijzer, A. Tettamanzi, P. Collet, J.I. van Hemert, M. Tomassini. Lecture Notes in Computer Science, vol. 3447 (Springer, Berlin, 2005), pp. 215–226
4. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (MIT Press, Cambridge, 1992)
5. P. Nordin, F. Francone, W. Banzhaf, Explicitly defined introns and destructive crossover in genetic programming, in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, Tahoe City, CA, 9 July 1995, ed. by J.P. Rosca, pp. 6–22
6. P.A. Whigham, Inductive bias and genetic programming, in *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, Sheffield, UK, 12–14 September 1995, ed. by A.M.S. Zalzala, vol. 414 (IEE, Piscataway, 1995), pp. 461–466

7. P.A. Whigham, Grammatically-based genetic programming, in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, Tahoe City, CA, 9 July 1995, ed. by J.P. Rosca, pp. 33–41
8. P.A. Whigham, Search bias, language bias, and genetic programming, in *Genetic Programming 1996: Proceedings of the First Annual Conference*, ed. by J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, Stanford University, CA, 28–31 July 1996 (MIT Press, Cambridge), pp. 230–237
9. H. Iba, Random tree generation for genetic programming. Technical Report ETL-TR-95-35, 14 November 1995 (ElectroTechnical Laboratory (ETL), Tsukuba, Japan, 1995)
10. W. Bohm, A. Geyer-Schulz, Exact uniform initialization for genetic programming, in *Foundations of Genetic Algorithms IV*, University of San Diego, San Diego, CA, 3–5 August 1996, ed. by R.K. Belew, M. Vose (Morgan Kaufmann, San Francisco, 1996), pp. 379–407
11. S. Luke, L. Panait, A survey and comparison of tree generation algorithms, in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, San Francisco, CA, 7–11 July 2001, ed. by L. Spector, E.D. Goodman, A. Wu, W.B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M.H. Garzon, E. Burke (Morgan Kaufmann, San Francisco, 2001), pp. 81–88
12. W.B. Langdon, Size fair and homologous tree genetic programming crossovers. Genet. Program. Evol. Mach. **1**(1/2), 95–119 (2000)
13. K. Chellapilla, Evolving computer programs without subtree crossover. *IEEE Trans. Evol. Comput.* **1**(3), 209–216 (1997)
14. S. Luke, Two fast tree-creation algorithms for genetic programming. *IEEE Trans. Evol. Comput.* **4**(3), 274–283 (2000)
15. M. Looks, On the behavioral diversity of random programs, in *GECCO '07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, London, 7–11 July 2007, vol. 2, ed. by D. Thierens, H.-G. Beyer, J. Bongard, J. Branke, J.A. Clark, D. Cliff, C.B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J.F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K.O. Stanley, T. Stutzle, R.A. Watson, I. Wegener (ACM Press, New York, 2007), pp. 1636–1642
16. S. Gustafson, E.K. Burke, G. Kendall, Sampling of unique structures and behaviours in genetic programming, in *Proceedings of the 7th European Conference on Genetic Programming, EuroGP 2004*, Coimbra, Portugal, 5–7 April 2004, ed. by M. Keijzer, U.-M. O'Reilly, S.M. Lucas, E. Costa, T. Soule. Lecture Notes in Computer Science, vol. 3003 (Springer-Verlag, Berlin, 2004), pp. 279–288
17. E.K. Burke, S. Gustafson, G. Kendall, Diversity in genetic programming: an analysis of measures and correlation with fitness. *IEEE Trans. Evol. Comput.* **8**(1), 47–62 (2004)
18. S. Gustafson, An analysis of diversity in genetic programming. PhD thesis, School of Computer Science and Information Technology, University of Nottingham, Nottingham, England, 2004)
19. R. Poli, W.B. Langdon, On the search properties of different crossover operators in genetic programming, in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, University of Wisconsin, Madison, WI, 22–25 July 1998, ed. by J.R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, R. Riolo (Morgan Kaufmann, San Francisco, 1998), pp. 293–301
20. U.-M. O'Reilly, F. Oppacher, Program search with a hierarchical variable length representation: genetic programming, simulated annealing and hill climbing, in *Parallel Problem Solving from Nature—PPSN III*, Jerusalem, 9–14 October 1994, ed. by Y. Davidor, H.-P. Schwefel, R. Manner. Lecture Notes in Computer Science, vol. 866 (Springer-Verlag, Berlin, 1994), pp. 397–406
21. W.B. Langdon, T. Soule, R. Poli, J.A. Foster, The evolution of size and shape, in *Advances in Genetic Programming 3*, Chap. 8, ed. by L. Spector, W.B. Langdon, U.-M. O'Reilly, P.J. Angeline (MIT Press, Cambridge, 1999), pp. 163–190
22. J.M. Daida, A.M. Hilss, Identifying structural mechanisms in standard genetic programming, in *Genetic and Evolutionary Computation—GECCO-2003*, Chicago, 12–16 July 2003, ed. by E. Cantú-Paz, J.A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M.A. Potter, A.C. Schultz, K. Dowsland, N. Jonoska, J. Miller. Lecture in Computer Science, vol. 2724 (Springer-Verlag, Berlin, 2003), pp. 1639–1651
23. S. Luke, Code growth is not caused by introns, in *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, Las Vegas, NV, 8 July 2000, ed. by D. Whitley, pp. 228–235
24. T. Soule, Exons and code growth in genetic programming, in *Proceedings of the 5th European Conference on Genetic Programming, EuroGP 2002*, Kinsale, Ireland, 3–5 April 2002, ed. by J.A. Foster, E. Lutton, J. Miller, C. Ryan, A.G.B. Tettamanzi. Lecture Notes in Computer Science, vol. 2278 (Springer-Verlag, Berlin, 2002), pp. 142–151

25. W. Banzhaf, W.B. Langdon, Some considerations on the reason for bloat. Genet. Program. Evol. Mach. **3**(1), 81–91 (2002)
26. T. Soule, R.B. Heckendorn, An analysis of the causes of code growth in genetic programming. Genet. Program. Evol. Mach. **3**(3), 283–309 (2002)
27. W. Banzhaf, P. Nordin, R.E. Keller, F.D. Francone, *Genetic Programming—An Introduction: On the Automatic Evolution of Computer Programs and its Applications* (Morgan Kaufmann, San Francisco, 1998)
28. R.M. Downing, Neutrality and gradualism: encouraging exploration and exploitation simultaneously with binary decision diagrams, in *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, Vancouver, Canada, 6–21 July 2006 (IEEE Press, Piscataway, 2006), pp. 615–622
29. W.B. Langdon, R. Poli, *Foundations of Genetic Programming* (Springer-Verlag, Berlin, 2002)
30. L. Beadle, Epoch X—Genetic Programming Analysis Software. http://www.epochx.com/epochx/default.asp, 2007–2008. Accessed 2 Mar 2009
31. F. Somenzi, Cudd: CU Decision Diagram Package release. http://vlsi.colorado.edu/ fabio/CUDD/, 1998. Accessed 2 Mar 2009
32. J. Whaley, JavaBDD. http://javabdd.sourceforge.net/, 2007. Accessed 2 Mar 2009
33. R.E. Bryant, Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.* **35**(8), 677–691 (1986)
34. S. Luke, Modification point depth and genome growth in genetic programming. Evol. Comput. **11**(1), 67–106 (2003)
35. S. Dignum, R. Poli, Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat, in *GECCO '07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, London, 7–11 July 2007, vol. 2, ed. by D. Thierens, H.-G. Beyer, J. Bongard, J. Branke, J.A. Clark, D. Cliff, C.B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J.F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K.O. Stanley, T. Stutzle, R.A. Watson, I. Wegener (ACM Press, New York, 2007), pp. 1588–1595
36. R. Poli, N.F. McPhee, General schema theory for genetic programming with subtree-swapping crossover: Part I. Evol. Comput. **11**(1), 53–66 (2003)
37. R. Poli, N.F. McPhee, General schema theory for genetic programming with subtree-swapping crossover: Part II. Evol. Comput. **11**(2), 169–206 (2003)
38. R.P. Salustowicz, J. Schmidhuber, Probabilistic incremental program evolution. Evol. Comput. **5**(2), 123–141 (1997)
39. N.F. McPhee, B. Ohs, T. Hutchison, Semantic building blocks in genetic programming, in *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, Naples, Italy, 26–28 March 2008, ed. by M. O'Neill, L. Vanneschi, S. Gustafson, A.I.E. Alcazar, I. De Falco, A.D. Cioppa, E. Tarantino. Lecture Notes in Computer Science, vol. 4971 (Springer, Berlin, 2008), pp. 134–145
40. L. Beadle, C.G. Johnson, Semantically driven crossover in genetic programming, in *Proceedings of the IEEE World Congress on Computational Intelligence*, Hong Kong, 1–6 June 2008 (IEEE, Piscataway, 2008), pp. 111–116
41. P.J. Angeline, Subtree crossover: building block engine or macromutation?, in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, Stanford, CA, 13–16 July 1997, ed. by J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba, R.L. Riolo (Morgan Kaufmann, San Francisco, 1997), pp. 9–17