Semantically Driven Crossover in Genetic Programming

Lawrence Beadle and Colin G. Johnson

***Abstract*—Crossover forms one of the core operations in genetic programming and has been the subject of many different investigations. We present a novel technique, based on semantic analysis of programs, which forces each crossover to make candidate programs take a new step in the behavioural search space. We demonstrate how this technique results in better performance and smaller solutions in two separate genetic programming experiments.**

***Index Terms*—Genetic programming, program semantics, crossover, reduced ordered binary decision diagrams.**

I. Introduction

This paper presents a novel algorithm (Semantically Driven Crossover—SDC) which is used to improve crossover in genetic programming (GP). The SDC algorithm has been developed based on analysis of the behavioural changes caused by crossover. The SDC algorithm works to enhance behavioural diversity in a GP run by not allowing child programs to be produced when they are semantically equivalent to the parent programs. We apply this technique to two boolean GP problems and demonstrate how it results in statistically significant improvements in performance, as well as a statistically significant decrease in code bloat.

The key feature of this concept is the use of a canonical representation of members of the population (reduced ordered binary decision diagrams—ROBDDs) to check for semantic equivalence without having to access the fitness function. This ability to check for equivalence provides us with an important tool to aid the search for a global optimum in the GP search space. This is because SDC eliminates crossovers that do not change the behaviour of candidate solution programs.

The remainder of this paper begins with a review of techniques that improve crossover and an introduction to ROBDDs. We describe the SDC algorithm and apply it to two different GP problems, comparing them with the traditional Koza standard crossover [1].

II. Improving Crossover and Effective Fitness

The approach we present simultaneously tackles two issues in GP. These are improving the performance of the crossover operation (in terms of the fitness measurement) and increasing the efficiency of the crossover operation (in terms of making a significant impact on reducing bloat). Therefore, in this section we review existing work in these areas.

Lawrence Beadle and Colin G. Johnson are with the Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK (email: {lb212, C.G.Johnson}@kent.ac.uk).

A. Improving Crossover in GP

When considering improving the crossover mechanism, there are three main approaches. These are: modifying the way in which parents are chosen for the crossover, choosing the swap points within the parent programs and using a system of pre and post crossover evaluation.

When choosing parents based on a particular characteristic, common sense would suggest that two parents with that characteristic should be selected more frequently in order to improve the performance of the GP run. This characteristic could be fitness based, or it could be due to another attribute (e.g. containing all the distinct terminals when it is prior knowledge that all terminals will be required to solve the problem). The danger of this process is that it could cause the GP to converge on a local optimum and may prevent it from making a large enough movement (because of the parsimony pressure of the parent selection method) in the search space to escape this local optimum.

Modifications to the method of choosing swap-points have been shown to have some effect on performance. The first major example of this is Koza standard crossover [1], which provides a 90% bias to choosing functions and 10% bias to choosing terminals as swap-points. The concept is that by putting a bias on the functions, it is more likely to cause a bigger movement in the search space. The disadvantage of such a technique is that later in the GP run it becomes more difficult to fine tune candidate solutions by changing a leaf of the tree. Rosca [2] proposed a similar idea based on a negative binomial distribution over tree depth. These experiments demonstrate a positive effect on GP performance; however, incorrect choice of swap-points could have a negative effect on performance. Another example of using the choice of swap points to influence crossover is homologous crossover [3], where sub trees which are structurally similar (defined using edit distances) are more likely to be crossed over.

Pre and post crossover evaluation compares the result of the crossover to the parents. An example of this is presented by O'Reilly [4], in which GP crossover is hybridised with two hill climbing techniques. Whilst this technique demonstrated that fewer fitness evaluations were required to reach the ideal solution, there is always the risk that the candidate solutions can be caught in local optima, as well as the additional computational requirement needed to implement this technique.

B. Efficiency in Crossover

One of the most studied side effects of crossover is bloat. Numerous authors [3], [5], [6], [7] have presented theories as to the cause of code growth; however, the cause of this growth remains a contentious issue. An important factor in terms of efficiency is whether all of the code included in a program is contributing to the fitness. Whilst numerous authors have studied introns [8], [9], [10], [11], [12], [13], [14] (code that does not contribute to the fitness of a program) there are three key aspects of introns that are dealt with in this paper.

In the first instance, there is the question of what actually makes up an intron. For the purposes of this paper, we will place introns into two categories (not the five categories described by Nordin [13]). These are unreachable code and redundant code. An example of unreachable code would be IF A1 D0 (IF A1 *(AND D0 D1)* D1). The emphasised segment is unreachable because of the condition of the nested IF statement. This represents an inefficiency because code is present but cannot be used.

The second issue is that of redundant code. An example of redundant code is AND A1 A1. This could just be reduced to A1. This is worse than unreachable code because we still have to process it to reach the answer, which represents a computational inefficiency. The ROBDD analysis we use in this paper reduces a program's behavioural representation, such that redundant and unreachable code is removed. This allows direct comparisons of the behaviour of the programs to be made without interference from introns.

The final and most important issue in terms of the mechanics of the crossover operator is the idea of *linkage*. Because of the random elements of the crossover operation, there is the potential to take two programs with 100% effective fitness, swap sub trees and, as a result, achieve either redundant, unreachable or both types of intron in the transplanted sub tree. Conversely, there is also the possibility of taking code from redundant or unreachable sub trees and transplanting them into a new parse tree such that they become effective and contribute to fitness.

A previous study by Luke [9] into controlling the crossover operation ensured that the swap points were not in unreachable sub trees. This study concluded that even when the crossover of unreachable code was controlled in this fashion, it still did not reduce code bloat. The aspect that is not considered by this study is whether the code is unreachable or redundant upon insertion to the receiving program. In this paper we compare pre and post crossover canonical representation of behaviour, in order to ascertain whether the insertion of the new code has caused a behavioural change, which would indicate that at least some of the inserted code has been effective.

III. Semantic Control Process

A. Measuring Semantic Equivalence

To enable us to analyse semantic characteristics of boolean programs we use our own Java implementation of GP, linked to the *Colorado University Decision Diagram Package* (http://vlsi.colorado.edu/~fabio/CUDD/) (CUDD) using the *JavaBDD* interface (http://javabdd.sourceforge.net/). The important functionality that this provides is the ability to reduce program representations by removing redundant and unreachable arguments. We can obtain canonical representations known as ROBDDs [15] of the behaviour of the boolean programs, which allows us to compare and analyse programs for semantic equivalence (see figure 1 for an example). Any two programs that reduce to the same ROBDD are semantically equivalent, and *vice versa*.

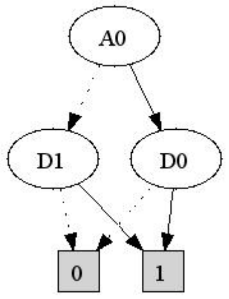

Figure 1. This example ROBDD is a canonical representation of behaviour, however, this behaviour could be represented by many different parse trees. Two examples of parse trees that would result in this behaviour are IF A0 D0 D1 and IF (NOT A0) D1 D0. In the diagram, circles represent variables (terminals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

B. Semantically Driven Crossover Algorithm

Our implementation of the SDC algorithm is a modification to Koza standard crossover [1]. We add the ability for the algorithm to check that the child programs of the crossover are semantically not equivalent to the parent programs. The SDC based crossover algorithm is presented below:

```
while ( number_of_programs < population_size ) {
  select p1 randomly (parent 1) (uniform distribution)
  select p2 randomly (parent 2) (uniform distribution)
  copy p1 into c1 (child 1)
  copy p2 into c2 (child 2)
  if ( random_number < crossover_probability )
  {
    choose swap_point1 on c1 (90% bias on functions)
    choose swap_point2 on c2 (90% bias on functions)
    perform crossover at swap points
    generate ROBDD of p1, p2, c1, c2
    if ( p1_ROBDD not equivalent to c1_ROBDD AND p2_ROBDD not equivalent to c1_ROBDD ) {
      add c1 to population
    }
    if ( p1_ROBDD not equivalent to c2_ROBDD AND p2_ROBDD not equivalent to c2_ROBDD ) {
      add c2 to population
    }
  } else {
    add c1 and c2 to population
```

```
        }
    }
```

Additionally, we check that we do not insert two members into the population when there is only one slot remaining in the population. The SDC algorithm essentially repeats Koza standard crossover until the child programs produced are not semantically equivalent to either parent. Whilst this does mean that there is an extra computational requirement in terms of both the numbers of crossovers to be performed, and the construction of the ROBDDs, this is offset by the increase in speed at fitness function evaluation, as the programs tend to be substantially smaller to process and score.

In a second experiment we implement crossover with only a state checker (SC) active such that if the child is equivalent to either parent, one of the parents would be added to the population instead of the child. This serves to demonstrate the effects of only stopping behaviourally neutral crossover and we will show how this has a significant effect on program size.

IV. Results

A. General GP Parameters

The general GP parameters we use in our experiments are 0.9 crossover, 0 mutation, 500 population size, 10% reproduction, max depth limit of 17, 50 generations in addition to a Ramped Half and Half (to depth 6) starting population and a 7 competitor tournament selection. We perform 100 runs of each experiment and all results reported are averaged over these 100 runs. With the exception of the 7 competitor tournament crossover, these parameters are used by Koza [1] as standard parameters. We use the 7 competitor tournament as it is quicker to execute than the fitness proportionate selection used by Koza.

B. Experiment 1: The 6 Bit Multiplexer

The 6 bit multiplexer has a semantic search space of size 2^{64}. The function set we use is {IF, AND, OR, NOT} and the terminal set is {A0, A1, D0, D1, D2, D3}. The objective is to map one of D0-D3 to the output, dependent on the two control bits A0 and A1 which are treated as a two bit binary number.

C. 6 Bit Multiplexer Results

Figure 2 shows that the application of a state checker in either SC or SDC form significantly reduces the depths of programs (confirmed using Paired T test at the 99% confidence level—The Paired T test is applied to compare the mean depth of programs over each generation for the different runs we present) when compared to standard crossover. Figure 2 also shows that the SDC average maximum score is significantly higher than the standard run (again confirmed using a Paired T test at the 99% confidence level).

Figure 3 shows the application of the state checker and the SDC algorithm increases the standard deviation of score significantly (confirmed using Paired T test at 99% level). The importance of this is that combined with the higher scores from Figure 2 the SDC algorithm is performing a wider search and yielding better results. In addition, it achieves this increased search using significantly smaller programs, which is a more efficient application of GP.

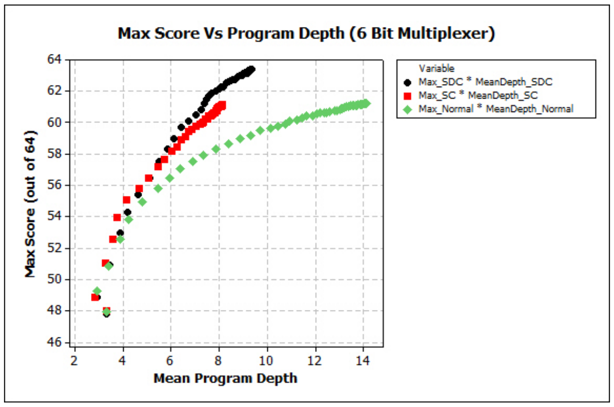

Figure 2. Mean maximum score plotted against mean program depth. Normal represents a standard GP run, SC represents a state checked run as described in III B and SDC represents a run using the SDC algorithm we describe.

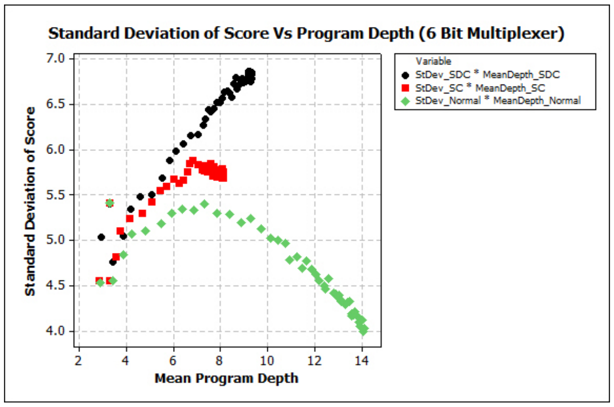

Figure 3. Mean standard deviation of score plotted against mean program depth. Normal represents a standard GP run, SC represents the use of the state checker and SDC represents the use of the SDC algorithm.

D. Experiment 2: Even 5 Parity

The even 5 parity problem has the function set {IF, AND, OR, NOT} and a terminal set {D0, D1, D2, D3, D4}. The objective is to output true when and only when an even number of inputs (D0-D4) are true.

E. Even 5 Parity Results

Figure 4 shows that the use of a state checker significantly reduces the depth of programs (confirmed using Paired T test at 99% confidence level). In addition to this, the score is significantly increased (confirmed using a Paired T test at 99% confidence level).

Figure 5 shows that the standard deviation is significantly increased (confirmed using Paired T test at 99% confidence level) when using the state checker and SDC algorithm. This again implies that a wider search of the behaviour space is taking place and, when this is combined with the decreased mean depths of the programs, a more efficient search (wider search with smaller programs) is being achieved.

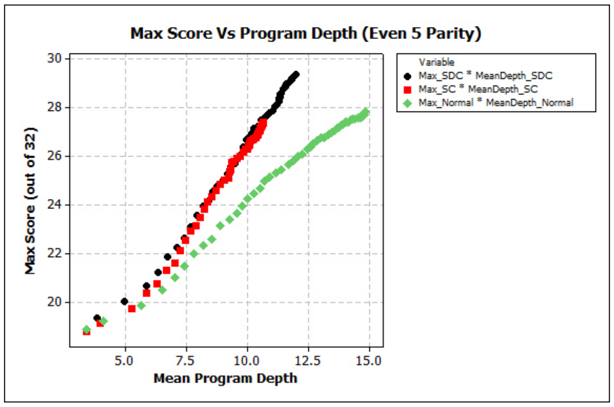

Figure 4. Mean maximum score plotted against mean program depth. Normal represents a standard GP run, SC represents a state checked run as described in 3.2 and SDC represents a run using the SDC algorithm we describe.

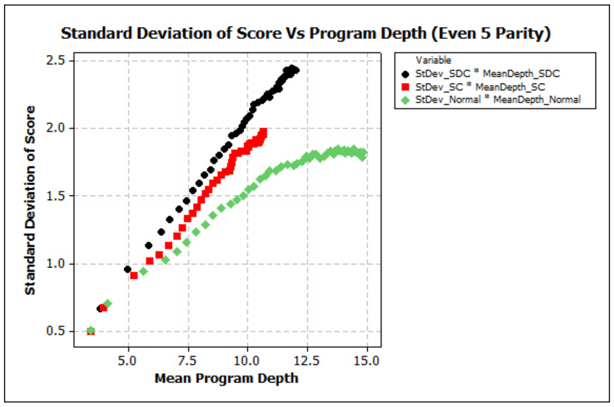

Figure 5. Mean standard deviation of score plotted against mean program depth. Normal represents a standard GP run, SC represents the use of the state checker and SDC represents the use of the SDC algorithm.

F. Results Summary

The results for all the experiments are summarized in table 1.

Table I
% IS THE PERCENTAGE OF RUNS THAT HAVE REACHED FULL SCORE BY GENERATION 50. DEPTH IS THE AVERAGE DEPTH OVER 100 RUNS. SCORE IS THE MEAN OF THE MAXIMUM SCORES OVER 100 RUNS. SCORE / DEPTH REPRESENTS A MEASURE OF EFFICIENCY IN THE AVERAGE CONTRIBUTION TO SCORE BY EACH LAYER OF PROGRAM CODE.

Experiment	Method	%	Score	Depth	Score/Depth
6 Bit Mux	SDC	85	63.40	9.333	6.793
	SC	44	61.17	8.131	7.523
	Standard	48	61.25	14.113	4.340
Even 5 Parity	SDC	5	29.34	12.016	2.442
	SC	1	27.39	10.660	2.569
	Standard	4	27.81	14.862	1.871

V. Discussion

The results we present in figures 2-5 and table 1 provide substantial and statistically significant support to show the SDC is able to improve results in GP runs. The effect of increasing the movement in the search space, despite the effect of destructive crossover, enables the SDC method to produce more successful runs where an ideal solution is found, and reaches a higher average maximum score for all the runs.

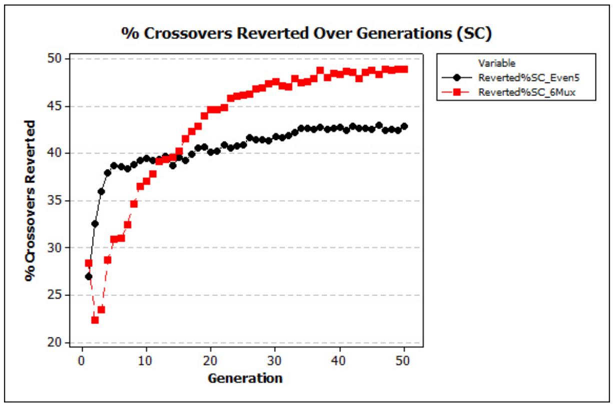

Figure 6. % of crossovers reverted over the generations using the state checker. This is data for the simplistic SC method applied to the 6 Bit Multiplexer and the Even 5 Parity problems. This result is averaged over 100 GP runs.

In addition to this, the fact that the standard deviation of scores is greatly increased (at statistically significant level) using the SDC adds weight to the argument that the SDC is performing a wider search and as a result attaining a wider variety of scores.

The application of the state checker functionality has made a significant impact on bloat. The average depth of programs is significantly reduced to approximately two thirds the average depth of programs in a normal run. Using our SC runs we are able to see explicitly what happens when the production of behaviourally neutral children is prevented. This supports the theory that introns do play at least some part in code bloat.

In addition to this we see a slight but statistically significant drop in score when the SC method is used, compared to the standard GP runs. This could be attributed to the opportunity to improve code that is lost when neutral but syntactically different crossovers occur.

In table 1 the figures for score divided by depth are presented. This gives an averaged and approximate guide as to how efficient a program is in terms of developing a high score with the smallest possible program size. Table 1 shows that the SC technique provides a higher score per depth of program, however, the score per depth of the SDC method also exceeds that of standard GP. This is the result of combination of two effects: the first is a wider search and therefore the possibility of finding more highly fit individuals; the second is the state checker's effect of reducing the size of the programs in such a way that at least some of the implanted code in the crossover is effective.

Figure 6 highlights an interesting fact when considering parent and child programs after crossover. For both the multiplexer and even parity problem, under normal GP run criteria, an increasing number of parents and children are behaviourally equivalent after crossover. This indicates that over 20%, and after the 10th generation, over 40%, of crossovers are not creating a movement in the behavioural search space. This is a worrying statistic as it shows that even in a best case scenario, over 40% of crossovers are wasted.

The SDC algorithm eliminates this effect and the increased scores attained represent the benefit of forcing each crossover to produce children which are behaviourally different to their parents.

VI. Application to Theory

There are a number of theories to explain the code bloat phenomenon in GP. We examine our experimental results (showing substantially reduced code bloat) and assess each theory in light of our results.

A. *Protection from Deletion*

The protection from deletion theory [3] states that introns protect valuable code during crossover. The reason that the valuable code needs to be protected is because crossover is mostly a destructive process and would destroy sub trees of useful code.

The SDC algorithm would magnify the effect of destructive crossover as it only checks whether a crossover is behaviourally neutral rather than whether it is constructive or destructive. As a result, if we prevent introns from being added to programs at the point of crossover, we would expect programs to perform worse in terms of overall score if we apply the protection from deletion theory. While we do see an increased range in scores (figures 3 and 5), we still see higher average maximum scores.

It is clear that allowing introns would serve to push up the scores of the worst performing programs and as a result increase the mean score, in the standard GP run. In terms of maximum score (which is the goal for many GP practitioners) the standard GP run results in worse performance when compared to the SDC in these experiments.

While we achieve a higher average maximum score using the SDC technique with a wider search, this raises the question of whether protection from deletion is valid. It could be argued that in the experiments we present it is not valid, because it would imply that introns are needed in order to achieve high scoring programs. In our experiments, whilst intron formation is explicitly prevented, the maximum scores outperform that of the standard GP technique.

Another problematic aspect of this theory is the idea that to achieve high scores GP evolution is dependent on an inefficient one to many mapping between behaviour and syntax respectively to achieve high scores. The system we have proposed in this paper has taken a step towards a one to one relationship between syntax and behaviour and outperforms the standard GP technique.

Our results indicate that protection from deletion may serve to increase the scores of the worse performing programs and, as a result, the mean score of the generation. The results we present disagree with the idea that introns are required to attain high performing programs.

B. *Fitness Causes Growth*

The theory fitness causes growth [16] can be split into two aspects. The first of these is the idea that programs may need to grow to attain higher scores as they may be missing essential parts. The second is the idea that a particular set of programs will drift to a region of the search space where they are more likely to attain a higher score and multiply resulting in the same standard behaviour.

Both of these factors are plausible under the system we present and although the SDC mean program size is substantially smaller than the standard GP, it does increase with score. Even with a canonical mapping of one to one between behaviour and syntax, both of the fitness causes growth situations are still possible.

One interesting difference between standard crossover and the SDC algorithm is the fact that whilst the search may move to areas of semantically equivalent programs, the lineage will be different because we force each crossover to make a new behavioural step. As a result, whereas in standard crossover, code may drift to semantically equivalent and fitness equivalent areas of the search space, in SDC crossover these clusters of programs may not be as large. This increased variance is confirmed by the substantial increase in standard deviation (figures 3 and 5) using the SDC algorithm.

C. *Removal Bias*

Removal bias [5] hinges on the idea that in the event of a fitness neutral crossover, it is more probable that a small sub tree will be swapped for a larger one during crossover. Behaviourally neutral crossovers are subsumed by fitness neutral crossovers, however, we see a substantial proportion of behaviourally neutral crossovers in figure 6. As a result of these experiments, the SDC algorithm serves to dampen the effect of removal bias, as it will prevent a substantial proportion of fitness neutral crossovers from occurring. Whilst it is possible for fitness neutral crossovers to occur under our system, the removal bias theory remains a plausible explanation for the code growth present in our experiments.

D. *Distribution of Program Sizes*

Dignum's recent work on GP theory in relation to bloat [7] suggests that a bias in the choice of crossover swap points (which is applicable to the 90% bias on functions, 10% on terminals crossover) which favours swapping larger sub trees into smaller programs will result in bloat. The SDC algorithm we present would have no effect on this process unless the bias caused behaviourally neutral crossovers. As a result of this, crossover bias may be one of the factors involved in increasing program sizes when SDC algorithm is applied.

VII. Conclusion

We have demonstrated that we can simultaneously and significantly increase performance of GP and decrease code bloat using the SDC algorithm on two separate GP problems. We have demonstrated that we can eliminate the addition of behaviourally neutral code at the point of crossover using ROBDDs. It is clear that, in the experiments we present, behaviourally neutral crossovers allow introns to propagate through the population increasing the average size of programs. From this we can deduce that code bloat is at least partially a result of intron creation through crossover.

The higher performance level is unsurprising as the SDC algorithm effectively forces programs to make more movements around the behavioural search space. Using the SDC algorithm, probability would dictate that the search is wider (which is supported by our results).

One factor that remains interesting is the high number of behaviourally neutral crossovers that occur under the standard GP process. Our simulation with only a semantic state checker running shows that between 20% and 50% of crossovers do not make a semantic movement around the search space. This represents a substantial waste in terms of computation and halves the power of the GP search.

Despite the theory that introns protect valuable code from destruction in crossover, the combination of two factors would indicate that introns are not required to achieve high performance in GP. These factors are increased performance and smaller program sizes. Based on these two factors, and the added computational burden of bigger programs, we can argue that introns are undesirable in GP.

VIII. Future Work

There are several areas of interest following on from this work. In the first instance there is the development of canonical representations to cater for other types of GP problem. Once these canonical representations are in place the SDC technique can be applied to other problems with relative ease. Secondly, there is the comparison of the SDC crossover techniques with other crossover techniques such as uniform crossover.

Another interesting experiment would be to attempt to run a GP with no introns present to compare the results with standard GP and current bloat theories. To do this we would require the ability to initialise populations without introns and control introns in crossover. In other work [17] we have developed a State Differential Algorithm, which generates semantically distinct programs with no introns. We could combine this with the SDC technique and a semantic pruning system to examine the effects of a GP with no introns.

One of the key aspects of whether a crossover will result in a behavioural change is how the code transplanted from the swap partner will work together with the rest of the program. A further understanding of linkage may yield clues as to how to make crossover more effective in terms of increasing the probability of changing the behaviour of a program with each crossover.

This paper has demonstrated the benefit of a wider, behaviourally driven search to GP. It would be useful to conduct an experiment which forced the GP to move into new areas of the search space as well as quantify how many times behaviours are produced to analyse and behavioural bias. To do this we could hybridize GP with a semantic based tabu search.

References

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

[2] J. P. Rosca and D. H. Ballard, "Rooted-tree schemata in genetic programming," in *Advances in Genetic Programming 3* (L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, eds.), ch. 11, pp. 243–271, Cambridge, MA, USA: MIT Press, June 1999.

[3] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. San Francisco, CA, USA: Morgan Kaufmann, Jan. 1998.

[4] U.-M. O'Reilly and F. Oppacher, "Hybridized crossover-based search techniques for program discovery," in *Proceedings of the 1995 World Conference on Evolutionary Computation*, vol. 2, (Perth, Australia), pp. 573–578, IEEE Press, 29 Nov. - 1 Dec. 1995.

[5] T. Soule and J. A. Foster, "Removal bias: a new cause of code growth in tree based evolutionary programming," in *1998 IEEE International Conference on Evolutionary Computation*, (Anchorage, Alaska, USA), pp. 781–186, IEEE Press, 5-9 May 1998.

[6] W. B. Langdon and R. Poli, "Fitness causes bloat: Mutation," in *Late Breaking Papers at the GP-97 Conference* (J. Koza, ed.), (Stanford, CA, USA), pp. 132–140, Stanford Bookstore, 13-16 July 1997.

[7] S. Dignum and 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* (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, and I. Wegener, eds.), vol. 2, (London), pp. 1588–1595, ACM Press, 7-11 July 2007.

[8] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster, "The evolution of size and shape," in *Advances in Genetic Programming 3* (L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, eds.), ch. 8, pp. 163–190, Cambridge, MA, USA: MIT Press, June 1999.

[9] S. Luke, "Code growth is not caused by introns," in *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference* (D. Whitley, ed.), (Las Vegas, Nevada, USA), pp. 228–235, 8 July 2000.

[10] T. Soule, J. A. Foster, and J. Dickinson, "Code growth in genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference* (J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, eds.), (Stanford University, CA, USA), pp. 215–223, MIT Press, 28–31 July 1996.

[11] T. Soule and J. A. Foster, "Code size and depth flows in genetic programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference* (J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, eds.), (Stanford University, CA, USA), pp. 313–320, Morgan Kaufmann, 13-16 July 1997.

[12] T. Soule and J. A. Foster, "Support for multiple causes of code growth in GP." Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97, 20 July 1997.

[13] P. Nordin, F. Francone, and 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* (J. P. Rosca, ed.), (Tahoe City, California, USA), pp. 6–22, 9 July 1995.

[14] K. Harries and P. W. H. Smith, "Code growth, explicitly defined introns and alternative selection schemes." www, 1998. Earlier version of Evolutionary Computation 6 (4), 336-360, 1998.

[15] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[16] W. B. Langdon and R. Poli, "Fitness causes bloat," in *Soft Computing in Engineering Design and Manufacturing* (P. K. Chawdhry, R. Roy, and R. K. Pant, eds.), pp. 13–22, Springer-Verlag London, 23-27 June 1997.

[17] L. Beadle and C. Johnson, "Behavioural diversity in genetic programming starting populations." Submitted for Publication.