Semantically Driven Mutation in Genetic Programming

Lawrence Beadle and Colin G Johnson

***Abstract*—Using semantic analysis, we present a technique known as semantically driven mutation which can explicitly detect and apply behavioural changes caused by the syntactic changes in programs that result from the mutation operation. Using semantically driven mutation, we demonstrate increased performance in genetic programming on seven benchmark genetic programming problems over two different domains.**

***Index Terms*—Genetic programming, program semantics, semantically driven mutation, reduced ordered binary decision diagrams.**

I. Introduction

In this paper we demonstrate the Semantically Driven Mutation (SDM) algorithm, which is used to improve the mutation operation in genetic programming (GP). The SDM algorithm has been developed based on semantic analysis of the changes caused by the mutation operator. The SDM algorithm works to improve performance by not allowing mutated programs to be produced when they are behaviourally equivalent to the original program. The aim of this is to avoid returning to sections of the search space that have effectively already been traversed. We compare the SDM algorithm to standard sub tree mutation in seven GP problems taken from the Boolean and artificial ant domains and demonstrate the superiority in performance produced by SDM.

In addition to the development of the SDM algorithm, we present results that combine this algorithm with our Semantically Driven Crossover (SDC) algorithm , in order to demonstrate the overall effects of semantically driven operators in GP. The key feature of the semantically driven operators is the ability to canonically represent candidate programs such that we can compare for the equivalence of behaviours.

In section II we review techniques to improve sub tree mutation. In section III we present the techniques we have used to abstract behaviours and our SDM algorithm. Section IV presents our results and section V presents a discussion of the results. In sections VI and VII, we present our conclusions and suggestions for future work respectively.

II. Literature Review

The SDM algorithm brings together two distinct areas of research. These are: the development of mutation techniques; and, the development of our ability to model the behaviour of programs. In section II-A we review several different mutation techniques and in section II-B we discuss techniques for modeling semantics in GP.

Lawrence Beadle and Colin G. Johnson are with the Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK (email: {L.Beadle-276, C.G.Johnson}@kent.ac.uk).

A. Mutation Techniques

Sub tree mutation selects a random point in a program tree and swaps the sub tree with another generated sub tree. In 1992 Koza [1] introduced sub tree mutation, but questioned the value of the operator (which was later demonstrated by Luke and Spector [2] to be comparable to crossover) and chose to perform most of his experiments without the mutation operator in use. Whilst the concept of sub tree mutation is relatively simple, there are more detailed practicalities that influence the relative performance of the GP with the use of the mutation operator.

The main variance is how different authors have constructed the new sub tree to replace the sub tree that was removed. Two examples of solutions to this are by Kinnear [3] and Langdon [4]. Kinnear created subtrees such that they could not increase the program depth by more than 15% after mutation. Langdon's size-fair sub tree mutation utilised a system which ensured that the new subtrees were on average the same size 50%-150% as the previously removed sub tree. A further variant of sub tree mutation is known as shrink mutation. In this system, a random sub tree is replaced by a terminal. Whilst Angeline [5] uses this type of mutation to aid his investigation into the sensitivity of the frequency of leaf selection in GP, he also shows that it helps to reduce program size.

Point mutation (or node replacement mutation) [6] picks a node and replaces it with a node of equivalent arity. This essentially simulates a single bit flip mutation from genetic algorithms. A similar idea is that of permutation, which selects a node and mutates the arguments of this node. Koza [1] used this technique in one experiment with little success. By contrast, Maxwell [7] had more success with a variant of permutation called swap.

Hoist mutation selects a sub tree from the program to be mutated and uses this sub tree to replace the full tree from which the sub tree was copied. Kinnear [3], [8] presented and made use of this technique with some success; but, this is potentially a highly destructive technique. Later research by McPhee and Hopper [9] indicated that specific patterns of code within successful programs can be traced back to very early programs in most GP runs. Mutation such as hoist may be highly destructive in that, if it were to alter the root of one of these common ancestors, it would cause a serious decrease in performance.

Whilst there exist a number of other techniques to mutate constants (some listed in Poli et al. [10, page 43]), the important point to understand in the context of our work is that these algorithms are processes to modify syntax, based on a selection of motivations (for example, to control program

978-1-4244-2959-2/09/$25.00

size). By contrast, our SDM algorithm is explicitly designed to cause a mutation that will result in change of behaviour. This sets it apart from other methods that merely process syntax.

B. Semantics in GP

A small number of studies have made use of a notion of program semantics to improve the design of aspects of GP.

In the related field of grammatical evolution, Majeed and Ryan [11] demonstrate a technique known as context aware mutation. The technique evaluates subtrees and works to prevent the mutation operator causing a destructive change in fitness. One of the limitations noted by the authors is the necessity of building up a repository of "good" subtrees to work with. Potentially, however, this technique could turn the standard mutation operator into another hill climbing operator. The danger of it becoming a hill climbing operator is that the fitness distribution across the search space may be rugged, increasing the possibility of premature convergence on a locally optimal solution compared to a non hill climbing algorithm.

Gustafson [12] developed two edit distances to sample semantic diversity in GP and conducted an analysis comparing behavioural diversity measures with changes in fitness. One of the limitations of the edit distance method is that it does not result in a canonical representation, which would be required by SDM to check for isomorphism of behaviours.

Semantic analysis methods are starting to appear in combination with crossover. McPhee et al. [13] used truth tables to analyse behavioural changes in crossover. A similar technique could be applied to mutation operators in order to assess the levels of behavioural change caused by a specific mutation. Whilst it is possible to represent behaviour using truth tables, a more efficient technique is that of using reduced ordered binary decision diagrams (ROBDDs) [14] to create reduced canonical representations to measure behavioural difference. Beadle and Johnson [15] used ROBDDs to compare pre and post crossover program states for semantic change. SDM is also based on ROBDDs, and is designed to apply to mutation ideas that have proven successful in crossover.

Other authors, such as Yanagiya [16] and Downing [17] have used *Binary Decision Diagrams* (BDDs) as a form of representation. The focus of Downing's work was to test the importance of neutrality in GP. In the case of Yanagiya, BDDs were used as a form of efficient representation to increase processing speed at the fitness function. Despite outlining special crossover and mutation processes to be used in the evolution of the BDDs, Yanagiya does not use BDDs to explicitly analyse behavioural states before and after operations. Our work is designed to test the effects of behavioural control at the point of the mutation operator.

III. Methods and Algorithms

The aim of this work is to demonstrate the positive effects of redesigning the mutation operator so that instead of merely altering syntax, it produces a guaranteed alteration of behaviour. In this section we present the problem domains we examine, our methods of abstraction and the pseudo code for the algorithm we use.

A. Test Problems Used

In our experiments we used seven test problems. These are the 6 and 11 bit multiplexer, even 4 and 7 parity, 5 and 9 majority and the artificial ant on the Santa Fe trail.

The objective of the 6 and 11 bit multiplexer problems is to interpret two or three (respectively) control bits as a binary number and choose the correct output bit based on the binary number. The fitness is the number of correct choices over all possible 64 or 2048 combinations of inputs for the 6 and 11 Boolean bits respectively. The function set is {IF, AND, OR, NOT} and the terminal set is {A0, A1, D0, D1, D2, D3}. 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 and 7 parity problems 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} for the even 4 parity and {D0, D1, D2, D3, D4, D5, D6} for the even 7 parity experiment.

The objective of the 5 and 9 majority problems 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} for the 5 majority problem and {D0, D1, D2, D3, D4, D5, D6 , D7, D8} for the 9 majority problem.

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 a full score. We use the benchmark *santa fe* trail [18] which represents 89 food pellets in a broken trail on a 32X32 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.

B. Abstraction

In order to measure semantic equivalence we developed a system to build canonical representations of behaviour of programs that evolve in our experiments.

For the multiplexer, even parity, and majority experiments we constructed a ROBDD [14] for each program and mutation of that program. The important functionality that this provides is the ability to reduce program representation by removing redundant and unreachable arguments. 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 example of an ROBDD can be found in figure 1.

We used three pieces of software to enable us to analyse the semantic representations of programs. We used a Java implementation of GP [19], linked to the *Colorado University*

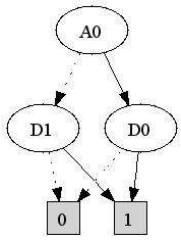


Fig. 1. This example ROBDD is a canonical representation of behavior. In the diagram, circles represent variables (terminals in the GP context); solid arrows represent true paths; dotted arrows represent false paths. The squares marked 1 and 0 represent output of true and false respectively. This behaviour 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.

Decision Diagram Package — CUDD [20] using the *JavaBDD* [21] interface.

For the artificial ant domain, we consider a behavioural model as a sequence of moves and orientations that represent the trail through which the ant has traveled during one execution of the ant control program (i.e. the GP candidate solution). When the artificial ant is simulated in GP we repeatedly execute the candidate solution until the ant has traveled 600 time steps [18], although, in the behavioural 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 (32X32) that contains no food pellets and we calculate the shape of the path for both the true and false branches of the IF-FOOD-AHEAD (if-then-else) function.

An example program for the artificial ant is as follows:

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

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

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

The character M represents one move and the characters N, S, E, W represent the orientations north, south, east and west respectively. The sub sequences within the set indicate when a branch of an IF-FOOD-AHEAD statement is being accessed and instruction sequence within those brackets indicates the path traveled during each branch of the condition. Because we are only concerned with modeling the shape of the trail, it is unimportant whether or not the ends of the IF-FOOD-AHEAD blocks have different orientations. Therefore, at the end of the conditions we reset the current orientation to the orientation before the ant entered the if branches.

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

$$\begin{aligned}
rep &::= \langle < expr > \rangle && (1)\\
expr &::= M|N|S|E|W && (2)\\
&\quad | < bracketExpr > && (3)\\
&\quad | < expr >, < expr > && (4)\\
bracketExpr &::= \langle < expr >, < expr > \rangle && (5)
\end{aligned}$$

In addition to this model structure, we condense the abstract representation in three ways. The first method is to remove duplicate sub branches of the same if statement and incorporate the paths as part of the fixed path the ant was on before the if statement. The second method 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 method 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.

C. SDM Algorithm

We present the pseudo code for SDM below:

```
for each program in population {
  if random_no < prob_mutation {
  counter = 0
  while counter < 5 {
      generate semantic_representation1 of
      original_program
      select mutation_point (uniform)
      generate sub tree using grow (depth 4)
      inset sub tree at mutation_point
      generate semantic_representation2 of
      mutated_program
      if semantic_representation1 ≡
      semantic_representation2 {
          revert mutated_program back to
          original_program
      } else {
          break
      }
      counter++
      }
  }
}
```

The SDM algorithm will try to mutate a program into a new behavioural state. The process involves performing a standard sub tree mutation; however, after each mutation attempt, the algorithm checks to see that each mutated program is semantically different to the original program. In some cases it may not be possible to semantically mutate a program (for example, if the program contained substantial inviable code) and as a result we have applied a counter system such that the mutation operator will have five attempts to behaviourally mutate a program, after which the original program is returned. Despite initial fears, this algorithm would be slow due to the creation of the representations of behaviour, run time appears roughly comparable.

IV. Results

A. Parameters

The parameters we use in our experiments are as follows: populations of 500 for 4 parity, 5 majority, 6 multiplexer

and the artificial ant (Santa Fe trail); populations of 4000 for the even 7 parity, 9 majority and 11 multiplexer; 10% elitist reproduction; 7 competitor tournament selection; ramped half and half (to depth 6) initialisation; depth 17 program size limit; 50 generation; 100 runs; 0.9 probability for standard and SDM mutation; and. when the SDC is used in addition to the SDM, we use 0.9 probability for both crossover and mutation.

B. SDM Results

Table I shows that the SDM algorithm significantly contributes to the performance of GP. For the overall maximum score (compared over all generations) standard mutation is always the worst performing mutation technique. In three of the experiments the SDM is the best performing method with regard to overall maximum scores. In the other four experiments, the combined SDC and SDM algorithms were the best performing GP runs.

When comparing the scores at generation 50, in all but one of the experiments standard sub tree mutation is the worst performing method. The combined SDC and SDM algorithms are always the highest performing; however, in three cases the SDM is statistically similar to the combined SDC and SDM algorithms. In the case of the artificial ant, all of the experiments produce a statistically similar result despite small variations in performance.

Unlike the SDC algorithm [15] (and further statistical data sheets at [19]), the relationship between SDM and its effects on the length of programs (shown in table II) appears to be problem dependent. It is clear that the combination of both the SDC and SDM algorithms does substantially increase code bloat. It is important to note that we used high values of 0.9 for both crossover and mutation in order to be consistent with our other experiments. Combinations of other crossover and mutation rates may well change this reading substantially, and this is an important topic for future research.

In summary, the results tell us, that whilst the SDM has no pronounced effect on bloat, it is clear that semantically driven mutation and the combination of semantically driven crossover and mutation do increase performance in GP.

C. Reverted Mutations

Figure 2 shows that there are a substantial number of state neutral mutations taking place and being reverted in both the 5 majority and artificial ant experiments at varying levels, and also shows that the combination of SDC and SDM produces more reversions, which has been confirmed statistically using a Paired T-test at the 95% confidence level. Figure 2 also illustrates an upwards trend over time in the numbers of reversions taking place in all of our experiments.

V. Discussion

Our results indicate that on some level the one to many relationship between behavioural and syntactic representations constitutes an inefficiency in performance. Using standard mutation, many fitness evaluations represent wasted computational effort, since the mutated program is semantically equivalent to the parent program. In every experiment, our overall comparisons demonstrated that the semantically driven operators must force increased levels of movement (as evidenced by the increased levels of reversion shown in figure 2) around the search space such that better solutions are found quicker compared to traditional sub tree mutation.

A speculative explanation for our varying program size results is the possibility that the increased search forced programs to move to different regions of the search space which required different numbers of nodes in the trees. This would be problem specific and as such we saw varying results in the program sizes produced by the SDM algorithm when compared to standard sub tree mutation.

In addition to the GP performance and program size results, we looked at the percentage of programs being reverted. One key characteristic of this was a positive trend over time in the level of reversions. It is well known that, as programs evolve, they increase in size ([22], [23], [24]), and with this increase in size, it is likely that there will be an increase in inviable areas of code [25], [26]. As such, it is less probable that mutation would operate on an active region of code, causing an increased number of program reversions because the SDM algorithm has to work harder to modify effective code.

A final point of note is that we have only applied SDM to a simple version of sub tree mutation. As mentioned in section II-A there are several other mutation techniques. Whilst each technique has a different mutation process, the SDM concept could be applied over the top of each of these different mutation processes. One fact to be drawn from increasing semantic diversity in standard sub tree mutation is the increase in overall performance noted in all of our experiments, and there is no reason to assume that applying the SDM concept on other mutation processes cannot demonstrate a similar increase in GP performance.

VI. Conclusion

In conclusion, there are four points to draw from this work. Firstly, semantically driven mutation statistically significantly increased the performance of GP in all seven experiments. Secondly, unlike semantically driven crossover, semantically driven mutation has no clear effect on the average size of the programs produced. Thirdly, the combination of semantically driven crossover and semantically driven mutation in our experiments did appear to bloat programs. This bloating may be due to the combination of crossover and mutation probability parameters we used in these experiments. Finally, the increase in the number of reversions over the generations indicates that the SDM algorithm has to work harder to continue to provide behavioural mutations as the generations increase.

VII. Future Work

The future avenues for this work can be divided into two research areas. Firstly, as the expandability of the SDM is limited by the ability to represent different problem domains, we can develop new representation models in order to see whether our results are sound in different contexts. The first possibility in this case would be extending our experiments

Problem	Operator	Overall Max (StDev)	PT Rank	G50 Max (StDev)	2T Rank	Success
Even 4 Parity	MUT	0.0970 (±0.0749)	3	0.0400 (±0.0483)	2	52% (G6)
	SDM	**0.0768 (±0.0774)**	**1**	**0.0231 (±0.0423)**	**1**	74% (G9)
	SDCSDM	0.0846 (±0.0833)	2	**0.0163 (±0.0303)**	**1**	76% (G7)
5 Majority	MUT	0.0527 (±0.0395)	3	0.0219 (±0.0241)	3	47% (G8)
	SDM	0.0389 (±0.0426)	2	0.0088 (±0.0154)	2	74% (G7)
	SDCSDM	**0.0375 (±0.0448)**	**1**	**0.0034 (±0.0108)**	**1**	90% (G11)
6 Multiplexer	MUT	0.0946 (±0.0520)	3	0.0511 (±0.0520)	3	42% (G8)
	SDM	0.0831 (±0.0553)	2	0.0406 (±0.0478)	2	47% (G7)
	SDCSDM	**0.0565 (±0.0642)**	**1**	**0.0091 (±0.0281)**	**1**	88% (G8)
Even 7 Parity	MUT	0.3584 (±0.0549)	3	0.2813 (±0.0339)	2	0% –
	SDM	**0.3511 (±0.0583)**	**1**	**0.2680 (±0.0373)**	**1**	0% –
	SDCSDM	0.3519 (±0.0569)	2	**0.2706 (±0.0304)**	**1**	0% –
9 Majority	MUT	0.1595 (±0.0340)	3	0.1221 (±0.0104)	2	0% –
	SDM	0.1576 (±0.0344)	2	0.1204 (±0.0098)	2	0% –
	SDCSDM	**0.1275 (±0.0462)**	**1**	**0.0769 (±0.0087)**	**1**	0% –
11 Multiplexer	MUT	0.2070 (±0.0620)	3	0.1334 (±0.0452)	3	0% –
	SDM	0.1942 (±0.0703)	2	0.1117 (±0.0380)	2	0% –
	SDCSDM	**0.1418 (±0.0944)**	**1**	**0.0384 (±0.0372)**	**1**	31% (G29)
Artificial Ant (SF)	MUT	0.3685 (±0.0719)	3	**0.3045 (±0.1202)**	**1**	5% (G13)
	SDM	**0.3411 (±0.0733)**	**1**	**0.2787 (±0.1315)**	**1**	10% (G8)
	SDCSDM	0.3466 (±0.0795)	2	**0.2764 (±0.1191)**	**1**	6% (G7)

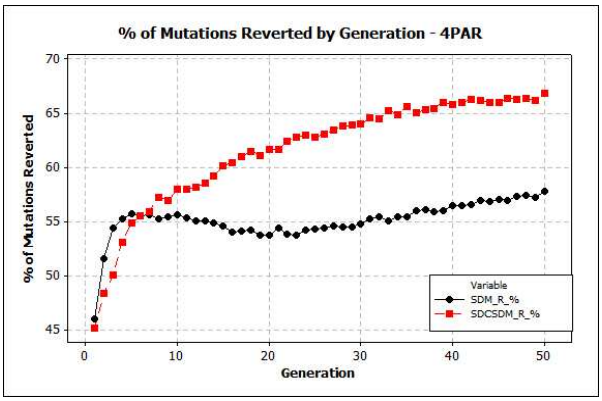
TABLE I

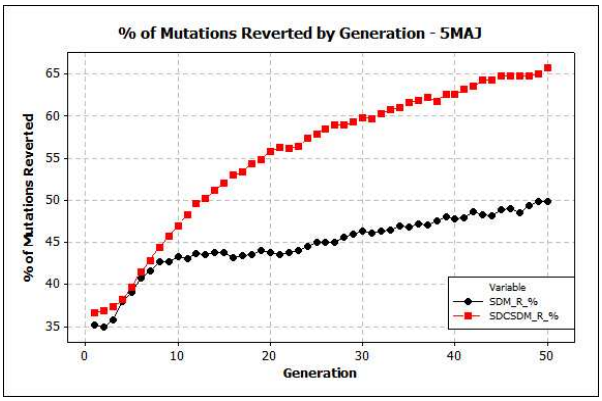
PROBLEM IS THE PROBLEM BEING EXAMINED. OPERATOR IS THE TYPE AND COMBINATION OF CROSSOVER AND MUTATION BEING USED WHERE MUT IS STANDARD MUTATION, SDM IS SEMANTICALLY DRIVEN MUTATION AND SDCSDM IS SEMANTICALLY DRIVEN CROSSOVER AND SEMANTICALLY DRIVEN MUTATION COMBINED. OVERALL MAX IS THE MEAN OF THE MAXIMUM SCORES OVER ALL GENERATIONS AND THE STDEV IS THE STANDARD DEVIATION OF THE MAXIMUM SCORES. THE SCORES ARE REPRESENTED IN STANDARDISED FITNESS (LOWER = BETTER). PT RANK IS THE PERFORMANCE RANK OF EACH OPERATOR BASED ON PAIRED T TESTS OF THE MAXIMUM SCORES OVER THE GENERATIONS. RANK 1 IS THE BEST PERFORMING AT THE 95% CONFIDENCE LEVEL. G50 MAX IS A MEAN OF THE MAXIMUM SCORES AT GENERATION 50 AND STDEV IS THE STANDARD DEVIATION OF THESE MAXIMUM SCORES. THE SCORES ARE REPRESENTED IN STANDARDISED FITNESS. 2T RANK IS THE PERFORMANCE RANKING BASED ON 2 SAMPLE T TESTS OF THE MAXIMUM SCORES AT GENERATION 50. AGAIN, RANK 1 IS THE BEST PERFORMING AT THE 95% CONFIDENCE LEVEL. SUCCESS IS THE NUMBER OF RUNS WHICH REACHED FULL SCORE AND THE BRACKET FIGURES REPRESENT THE FIRST GENERATION IN ANY RUN AT WHICH FULL SCORE WAS ACHIEVED.

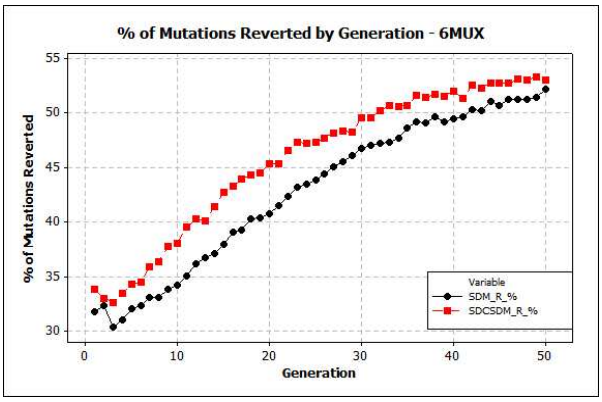
Problem	Operator	Overall Length (StDev)	PT Rank
Even 4 Parity	MUT	159.08 (±39.87)	2
	SDM	**144.62 (±29.45)**	**1**
	SDCSDM	160.44 (±51.69)	2
5 Majority	MUT	90.32 (±28.82)	2
	SDM	**82.27 (±21.75)**	**1**
	SDCSDM	113.85 (±50.39)	3
6 Multiplexer	MUT	**51.95 (±16.08)**	**1**
	SDM	**52.29 (±12.85)**	**1**
	SDCSDM	72.07 (±24.62)	2
Even 7 Parity	MUT	**206.04 (±44.67)**	**1**
	SDM	215.71 (±47.86)	2
	SDCSDM	339.61 (±131.06)	3
9 Majority	MUT	**80.34 (±23.19)**	**1**
	SDM	86.78 (±25.30)	2
	SDCSDM	307.47 (±183.12)	3
11 Multiplexer	MUT	**38.92 (±9.64)**	**1**
	SDM	41.02 (±8.87)	2
	SDCSDM	112.49 (±49.60)	3
Artificial Ant (SF)	MUT	**111.65 (±26.02)**	**1**
	SDM	121.14 (±28.66)	2
	SDCSDM	190.21 (±38.32)	3

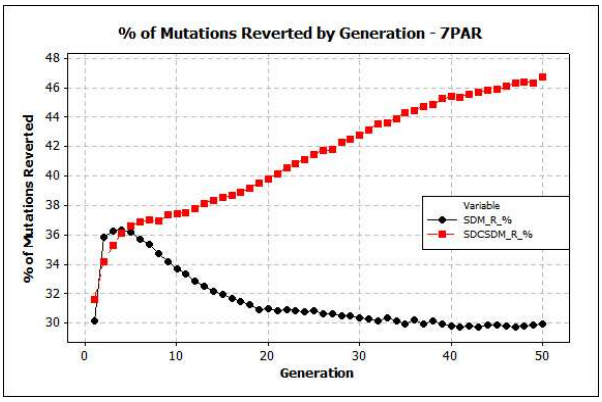
TABLE II

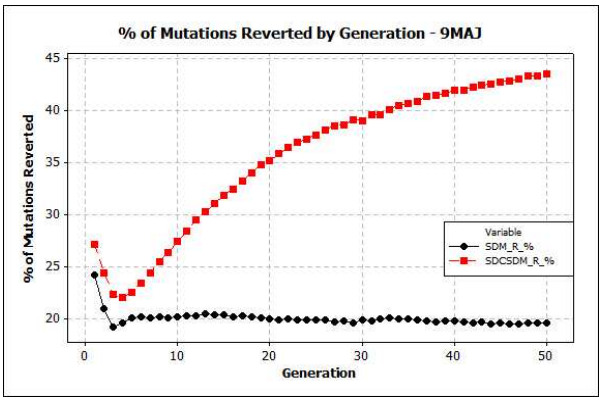
PROBLEM IS THE PROBLEM BEING EXAMINED. OPERATOR IS THE TYPE AND COMBINATION OF CROSSOVER AND MUTATION BEING USED WHERE MUT IS STANDARD MUTATION, SDM IS SEMANTICALLY DRIVEN MUTATION AND SDCSDM IS SEMANTICALLY DRIVEN CROSSOVER AND SEMANTICALLY DRIVEN MUTATION COMBINED. OVERALL LENGTH IS AN AVERAGE OF THE AVERAGE PROGRAM LENGTHS FOR ALL GENERATIONS AND STDEV REPRESENTS THE STANDARD DEVIATIONS OF THESE MEAN LENGTHS. PT RANK INDICATES THE RANK OF THE INDIVIDUALS LENGTH (SHORTEST BEING BEST) USING A PAIRED T-TEST ANALYSED AT THE 95% CONFIDENCE LEVEL.

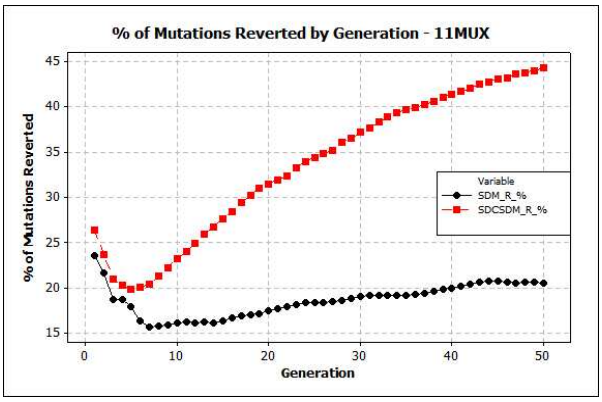












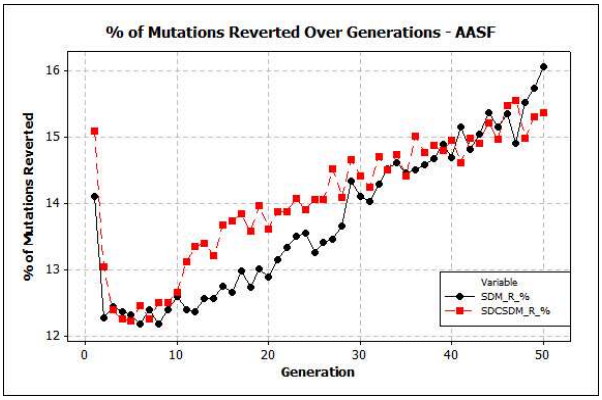


Fig. 2. The seven graphs indicate the percentage of mutations that were reverted in 2 of our experiments. These are the 5 majority experiment (5MAJ) and the artificial ant experiment (AASF). SDM_R_% indicates the reversion rate for the single SDM experiments and SDCSDM_R_% indicates the rate of reversions for the combined SDC and SDM runs. All results are averaged over 100 runs.

to cover a regression style domain. In order to expand SDM into other domains, we require the development of a canonical representation technique for that domain. In the case of symbolic regression, it may be possible to use zero-suppressed BDDs [27] as a viable representation.

Secondly, we can work towards building a semantically driven GP to bridge the inefficiency gap caused by the one to many relationships scenario between behaviours and syntax. A combination of semantic initialisation, crossover, and mutation, would allow us to evolve a more behaviourally diverse range of programs and, we anticipate, to reach better solutions in less time than standard GP.

REFERENCES

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

[2] S. Luke and L. Spector, "A revised comparison of crossover and mutation in genetic programming," in *Genetic Programming 1998: Proceedings of the Third Annual Conference* (J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, eds.), (University of Wisconsin, Madison, Wisconsin, USA), pp. 208–213, Morgan Kaufmann, 22-25 July 1998.

[3] K. E. Kinnear, Jr., "Evolving a sort: Lessons in genetic programming," in *Proceedings of the 1993 International Conference on Neural Networks*, vol. 2, (San Francisco, USA), pp. 881–888, IEEE Press, 28 Mar.-1 Apr. 1993.

[4] W. B. Langdon, "The evolution of size in variable length representations," in *1998 IEEE International Conference on Evolutionary Computation*, (Anchorage, Alaska, USA), pp. 633–638, IEEE Press, 5-9 May 1998.

[5] P. J. Angeline, "An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover," 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. 21–29, MIT Press, 28–31 July 1996.

[6] B. McKay, M. J. Willis, and G. W. Barton, "Using a tree structured genetic algorithm to perform symbolic regression," in *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA* (A. M. S. Zalzala, ed.), vol. 414, (Sheffield, UK), pp. 487–492, IEE, 12-14 Sept. 1995.

[7] S. R. Maxwell, "Why might some problems be difficult for genetic programming to find solutions?," in *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996* (J. R. Koza, ed.), (Stanford University, CA, USA), pp. 125–128, Stanford Bookstore, 28–31 July 1996.

[8] K. E. Kinnear, Jr., "Fitness landscapes and difficulty in genetic programming," in *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*, vol. 1, (Orlando, Florida, USA), pp. 142–147, IEEE Press, 27-29 June 1994.

[9] N. F. McPhee and N. J. Hopper, "Analysis of genetic diversity through population history," in *Proceedings of the Genetic and Evolutionary Computation Conference* (W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, eds.), vol. 2, (Orlando, Florida, USA), pp. 1112–1120, Morgan Kaufmann, 13-17 July 1999.

[10] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[11] H. Majeed and C. Ryan, "Context-aware mutation: a modular, context aware mutation operator for genetic programming," 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. 1651–1658, ACM Press, 7-11 July 2007.

[12] S. Gustafson, *An Analysis of Diversity in Genetic Programming*. PhD thesis, School of Computer Science and Information Technology, University of Nottingham, Nottingham, England, Feb. 2004.

[13] N. F. McPhee, B. Ohs, and T. Hutchison, "Semantic building blocks in genetic programming," in *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008* (M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, eds.), vol. 4971 of *Lecture Notes in Computer Science*, (Naples), pp. 134–145, Springer, 26-28 Mar. 2008.

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

[15] L. Beadle and C. Johnson, "Semantically driven crossover in genetic programming," in *Proceedings of the IEEE World Congress on Computational Intelligence*, (Hong Kong), pp. 111–116, IEEE, 1-6 June 2008.

[16] M. Yangiya, "Efficient genetic programming based on binary decision diagrams," in *1995 IEEE Conference on Evolutionary Computation*, vol. 1, (Perth, Australia), pp. 234–239, IEEE Press, 29 Nov. - 1 Dec. 1995.

[17] R. M. Downing, "Evolving binary decision diagrams using implicit neutrality," in *Proceedings of the 2005 IEEE Congress on Evolutionary Computation* (D. Corne, Z. Michalewicz, M. Dorigo, G. Eiben, D. Fogel, C. Fonseca, G. Greenwood, T. K. Chen, G. Raidl, A. Zalzala, S. Lucas, B. Paechter, J. Willies, J. J. M. Guervos, E. Eberbach, B. McKay, A. Channon, A. Tiwari, L. G. Volkert, D. Ashlock, and M. Schoenauer, eds.), vol. 3, (Edinburgh, UK), pp. 2107–2113, IEEE Press, 2-5 Sept. 2005.

[18] W. B. Langdon and R. Poli, *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[19] L. Beadle, "Epoch X - Genetic Programming Analysis Software." http://www.epochx.com/epochx/default.asp, 2007-2008.

[20] F. Somenzi, "Cudd: CU Decision Diagram Package release." http://vlsi.colorado.edu/ fabio/CUDD/, 1998.

[21] J. Whaley, "JavaBDD." http://javabdd.sourceforge.net/, 2007.

[22] R. Poli, W. B. Langdon, and S. Dignum, "On the limiting distribution of program sizes in tree-based genetic programming," Technical Report CSM-464, Department of Computer Science, University of Essex, Dec. 2006.

[23] S. Luke, "Modification point depth and genome growth in genetic programming," *Evolutionary Computation*, vol. 11, pp. 67–106, Spring 2003.

[24] T. Soule and R. B. Heckendorn, "An analysis of the causes of code growth in genetic programming," *Genetic Programming and Evolvable Machines*, vol. 3, pp. 283–309, Sept. 2002.

[25] 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.

[26] 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.

[27] S. Minato, "Implicit manipulation of polynomials using zero-suppressed bdds," in *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, (Washington, DC, USA), p. 449, IEEE Computer Society, 1995.