Towards Effective Semantic Operators for Program Synthesis in Genetic Programming

Stefan Forstenlechner
Natural Computing Research & Applications Group
School of Business
University College Dublin
Dublin, Ireland
stefan.forstenlechner@ucdconnect.ie

David Fagan
Natural Computing Research & Applications Group
School of Business
University College Dublin
Dublin, Ireland
david.fagan@ucd.ie

Miguel Nicolau
Natural Computing Research & Applications Group
School of Business
University College Dublin
Dublin, Ireland
miguel.nicolau@ucd.ie

Michael O'Neill
Natural Computing Research & Applications Group
School of Business
University College Dublin
Dublin, Ireland
m.oneill@ucd.ie

ABSTRACT

The use of semantic information in genetic programming operators has shown major improvements in recent years, especially in the regression and boolean domain. As semantic information is domain specific, using it in other areas poses certain problems. Semantic operators require being adapted for the problem domain they are applied to. An attempt to create a semantic crossover for program synthesis has been made with rather limited success, but the results have provided insights about using semantics in program synthesis. Based on this initial attempt, this paper presents an improved version of semantic operators for program synthesis, which contains a small but significant change to the overall functionality, as well as a novel measure for the comparison of the semantics of subtrees. The results show that the improved semantic crossover is superior to the previous semantic operator in the program synthesis domain.

CCS CONCEPTS

• **Computing methodologies → Genetic programming**;

KEYWORDS

Genetic Programming, Program Synthesis, Semantics, Operators

ACM Reference Format:
Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2018. Towards Effective Semantic Operators for Program Synthesis in Genetic Programming. In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205455.3205592

1 INTRODUCTION

Semantic information has been used in Genetic Programming (GP) to develop more advanced operators that, instead of doing random syntactic changes, incorporate the behaviour of an individual into deciding how to adapt and improve a solution. Most research in the area of semantics and GP was performed in the boolean and regression domains [1, 10, 15, 17] and has proven that semantic operators achieve better performance than operators that only work on a syntactic level. As semantics are dependent on the problem domain, an operator cannot be applied to another domain without changes, such as adapting measures to compare semantics between subtrees. Recently, steps were taken to harness the benefits of semantics in the domain of program synthesis [5]. Whereas the boolean domain and regression domain only operate on a single data type, boolean and numerical values respectively, program synthesis operates on a range of different data types at the same time and even data structures. Therefore measuring, for example, semantic similarity is a more complex task. Although the crossover operator in [5] has given interesting insights about semantics in the program synthesis domain, several shortcomings of the approach used in that paper have been identified as well. To further improve the performance of GP operators in program synthesis, this paper strives towards more effective semantic operators by addressing these shortcomings.

The rest of the paper is structured in the following way. Section 2 gives an overview of semantics and semantic operators, especially how semantics have been used in program synthesis so far. Section 3 describes in detail how new effective semantic operators have been created that address previous shortcomings. Section 4 outlines the experimental setup, which is discussed in Section 5. Conclusion and future work are discussed in Section 6.

2 RELATED WORK

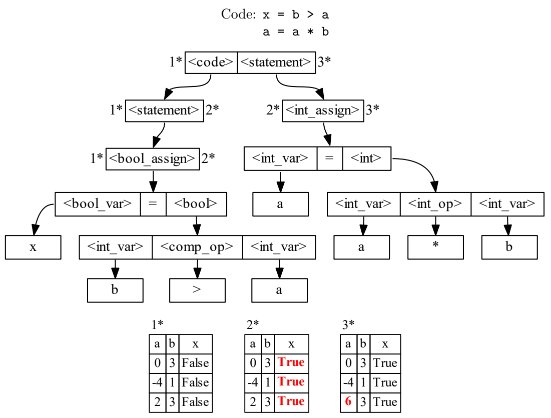
In this section, semantics and some semantic operators introduced to GP are outlined, as well as a detailed description of the Semantic Crossover for Program Synthesis (SCPS) [5] is given, which is going to be improved in this paper.

2.1 Semantics

Semantics can be defined as "the behavior of a program, once it is executed on a set of data" [17]. In the regression domain, a program is an arithmetic expression, which returns a vector of real values, when executed on data. Similarly, in the boolean domain, the semantics is a vector of boolean values. This semantic information can then be used in diverse ways to improve the performance of GP. Most approaches are based on new crossover [1, 10, 12, 14, 15] or mutation [2, 13] operators, but also semantic selection operators have been created [6, 7]. The downside of semantic operators, in most cases, is that as semantics are problem domain specific, operators based on semantics are domain specific as well. In contrast, conventional GP operators operate on a syntactic level, hence they can be used regardless of the problem domain.

Two important properties of semantics that contribute to performance improvements are semantic diversity and locality [15, 17]. While a high semantic diversity is necessary for covering the search space, semantic locality, which means a small change in a program corresponds to a small change in semantics and therefore fitness, is essential for the search performance [15].

A more direct approach of applying semantics has been introduced with Geometric Semantic Genetic Programming (GSGP) [11]. On the one hand, GSGP uses tailored semantic operators to guarantee that solutions incrementally become better or at least are not able to become worse. On the other hand, this approach has multiple limitations, e.g. it is restricted to a small range of problem domains and solutions increase in size rapidly if no operator is used to simplify the solutions. A complete survey about semantics and how semantics has been applied is available in [17].


Code:
```
x = b > a
a = a * b
```

1*

a	b	x
0	3	False
-4	1	False
2	3	False

2*

a	b	x
0	3	**True**
-4	1	**True**
2	3	**True**

3*

a	b	x
0	3	True
-4	1	True
6	3	True

Figure 1: A sample of a code with its corresponding derivation tree and its trace for three different inputs. The state of the variables before running the code is shown in variable setting 1*. While the variable setting 3* shows the state of the variables after executing code, 2* displays the intermediate state after executing the first statement and before the second one. The numbers 1*-3* are also shown within the derivation tree to indicate the semantics before and after executing a certain node.

2.2 Semantics with Traces in Program Synthesis

Semantics being domain specific means we need to specify semantics for each domain and also tweak the semantic operators. For program synthesis, Forstenlechner et al. [5] have used the trace of a program as its semantics, as traces describe the behaviour of a program. A trace logs variable state changes of the execution of a program. Similar to a GP solution in regression, which produces a vector of real values as output for the whole solution but also for subtrees, traces are available for all subtrees that are executable. This is important as many semantic operators exchange subtrees of a GP tree and need to be able to evaluate semantics of subtrees instead of a whole solution.

A short example of a program, the corresponding GP tree and its trace is shown in Figure 1, which is used to explain how the semantics for program synthesis has been defined in [5]. The figure shows a short program that only consists of two statements. The GP tree is a derivation tree that is generated when using a grammar-based variant of GP. The bottom of the figure shows the variable settings at different states during the execution of the program referenced with numbers from 1⋆ to 3⋆. It should be noted that in contrast to regression or boolean solutions, the output may not only be a single vector of some values but actually multiple vectors, as a program contains multiple variables that undergo state changes. The variable setting 1⋆ contains the initial setting, which might be the training data. In this example, it consists of only three cases. The variable setting 2⋆ shows the state of the variables after executing the first line of the code, but before executing the second line. 2⋆ is the semantic output of the subtree to the left of the root node. The variable setting 3⋆ contains the state of the variable after executing both lines of code and is the semantics of the whole tree. Depending on the statement that is executed a variable can change its stored value in all, some or even no cases. Variable x is changed in all instances after executing the first statement, while variable a has only been changed in one of three cases.

1⋆ to 3⋆ are also shown within the GP tree to indicate at which point a certain variable state is established. A number to the left indicates the variable setting before and to the right after executing the node. Not every node has a number with a variable setting. To evaluate the semantic change when exchanging a subtree on the lower levels that cannot be executed on its own, the first parent node that can be executed is used.

2.3 Semantic Operators

A variety of semantic operators has been introduced to GP and have shown to improve performance compared to conventional operators. This study focuses on crossover and mutation, but also selection operators are able to harness semantic information [6, 7].

As stated previously most research around semantics has been conducted in the regression and boolean problem domain. A series of operators have been introduced, some of which have been adapted and improved over time [12, 14]. One of which leads to the Most Semantic Similarity based Crossover (MSSC) [15] in the regression domain. This crossover operator selects multiple pairs of subtrees, one from each parent, and chooses the pair that is most semantically similar. The most semantically similar pair is the one that has semantics, real-valued output vectors, whose mean absolute difference is smallest compared to the other pairs. At the same time, the semantics of a subtree pair is not allowed to be equivalent, because if they would be equivalent, the change would have no impact on the overall fitness.

Most semantic operators are based on the principle that the change that is being made has to change the semantics and if possible should be similar at the same time. Semantic mutation operators were created in a similar way, but instead of selecting a subtree from a second parent, it was generated at random, as in conventional mutation operators [2, 13].

2.4 Semantic Crossover for Program Synthesis

A step towards using semantic information in operators within the program synthesis domain was taken by Forstenlechner et al. [5] by introducing semantic crossover for program synthesis (SCPS). This operator is based on MSSC by Nguyen et al. [15]. Due to the difference in problem domains, not only a different semantic measure had to be used, but also the crossover itself was adapted to fit the needs of the problem domain and semantic measure better. The pseudocode of the semantic crossover for program synthesis is shown in Algorithm 1.

The main difference, and maybe the main drawback, is how the overall crossover process has been changed from MSSC to SCPS. MSSC chooses *Max_Tries* pairs of subtrees from both parents, the semantics of the subtree pairs are compared, and the most semantically similar pair is chosen for crossover. SCPS chooses one subtree from the first parent and multiple from the second parent. Then it compares the semantics of all subtrees selected from the second parent with the one from the first parent. Finally, SCPS chooses the one that is most semantically similar. Even the semantic measure only uses a single type of variable for the semantic comparison. Variables with other data types were not part of the comparison. The reason was that multiple different distance measures have been used depending on the type of variable compared (e.g. Levenshtein distance for string, Hamming distance for bool). Therefore, the semantic similarity or difference for a pair of subtrees can hardly be compared to a different pair of subtrees. That lead to the drawback that, if the subtree selected from the first parent was at a position within a statement that had no effect on the semantics of the statement, no matter what it is replaced with the semantics would not change. This lead to SCPS having to choose one of the subtrees selected from the second parent at random and was not regarded as a semantic crossover, but rather a random crossover. According to the results in [5], the percentage of this fallback to random crossover occurring was rather high on all problems tackled and even up to around 50% on one problem.

Another conclusion that was drawn from [5] is that the semantic similarity measure used in SCPS was rather complex. Different similarity measures were proposed, one for each data type. In a semantic comparison, each variable from the two traces was compared with the according similarity measure to get a value for each variable that indicates the semantic similarity. These values were then used to find the subtree from the second parent that was most semantically similar to the first one but not equivalent. This seems to be a rather complex procedure considering the high amount of

Algorithm 1 Semantic Crossover for Program Synthesis (SCPS) [5]

```
select crossover point from first parent
select Max_Tries possible subtrees from second parent
if no subtrees of same type as crossover point available then
    return do nothing
end if
get semantics of every selected subtree from second parent
calculate semantic differences for every selected subtree per type
if differences then
    select random type
    select most semantically similar subtree based on selected type
else
    select random subtree for crossover from second parent
end if
crossover with selected subtree
```

crossover operations not even finding a single subtree that is semantically different. Forstenlechner et al. even stated in the future work section, that a simpler check might actually suffice.

3 EFFECTIVE SEMANTIC OPERATORS FOR PROGRAM SYNTHESIS

In this section, we present novel semantic operators which use semantic information more effectively as previously presented operators using insights about their shortcomings and addressing them. A semantic crossover is described in the next section, followed by a mutation operator that acts on a similar principle.

3.1 Effective Semantic Crossover for Program Synthesis

An adapted version of SCPS [5] that fixes the issues discussed in Section 2.4 has been created and named Effective Semantic Crossover for Program Synthesis (ESCPS). Pseudocode for ESCPS is shown in Algorithm 2. The first small but important change is that similar to MSSC a pair of subtrees is selected. One subtree from each parent. The semantic information from the subtree of the first parent has already been collected during the fitness evaluation, so no further overhead is required for that part.

Algorithm 2 Effective Semantic Crossover for Program Synthesis (ESCPS)

```
repeat
    select subtree from first parent
    select subtree of matching type from second parent
    calculate semantics of second subtree
    compare semantics for partial change
    if partial change found? then
        do crossover between subtrees
        return
    end if
until successful crossover or maximum tries
check all subtree pairs again for any difference
do crossover with the first pair that shows any difference
```

To be able to compare the semantics of the two subtrees, the semantics of the subtree from the second parent has to be evaluated based on the variable setting before executing the subtree from the first parent. This ensures that both subtrees have been executed on the same state of variables. It is important to note that this is not a fitness evaluation. The pseudocode that describes the process of establishing the semantics of the subtree from the second parent, which has also been used for SCPS, is shown in Algorithm 3.

Algorithm 3 Calculate semantics for a subtree from the second parent

```
input1, output1 ← semantics of subtree from first parent
set variables to input1
output2 ← execute subtree from second parent
```

In the next step, the semantics of the two subtrees are compared. As explained in Section 2.4, the previous measure for similarity was rather complex and still was unable to find many subtrees that produced different semantics. Therefore, a simpler semantic measure that is easier to use and implement for various data types has been established that checks for a *partial change*. As the semantics of a subtree is a vector of values for each variable, the measure checks for every variable if there is at least one difference between the semantics from the subtrees in a single entry in the vector, but the vectors are not allowed to be completely different. Or to put it in other words, at least one entry has to be different and at least one entry has to be identical. This provides the information that the subtrees are not equivalent but have some similarity.

If a partial change has been found, the two subtrees are used for crossover. No further steps need to take place. This can reduce the number of semantic comparisons that need to take place compared to SCPS and even MSSC, which reduces the computational effort. Although in the worst-case scenario the number of semantic comparisons will be identical to SCPS and MSSC. As it is still possible to not find a partial change after a maximum number of tries and to avoid falling back to random crossover right away, an additional second semantic measure is used. The second semantic measure checks for *any change* in the semantics between two subtrees. As multiple pairs of subtrees and the corresponding semantics have already been calculated. The same subtrees are checked with the second measure and the first pair of subtrees that shows any difference is selected for crossover. The intention is to avoid falling back to random crossover and use the semantic information gathered in the previous loop.

Only if both semantic measures fail to find a partial or any change, crossover falls back to the default behaviour, which is selecting subtrees at random. So, one subtree pair that has been selected within the loop is used for crossover.

3.2 Effective Semantic Mutation for Program Synthesis

As semantics can be used in an equivalent way in mutation as it is used in crossover, an Effective Semantic Mutation for Program Synthesis (ESMPS) operator has been created as well. It works on the same principle as the ESCPS described above. Like conventional subtree mutation, a new random subtree is generated, but the semantics of the new subtree is evaluated to decide if it should be used. Again, a maximum number of tries can be set to do so and first ESMPS checks for a partial change as long as the maximum number of tries has not been exceeded. Afterwards, it falls back to check for any change, before it has no other choice than to fall back to random mutation. The Pseudocode would be very similar to Algorithm 2, except the third line would be replaced with "generating a random subtree", which would be used in line four instead of the "second subtree".

Table 1: Experimental parameter settings

Parameter	Setting
Runs	100
Generations	300
Population size	1000
Selection	Lexicase
Crossover probability	0.9
Mutation probability	0.05
Elite size	1
Node limit	250
Variables per type	3
Max execution time	1 second
Max_Tries	10

4 EXPERIMENTAL SETUP

The goal of the experiments is to show that semantic operators are able to outperform conventional operators even in the program synthesis domain as well as that the semantic operators do not have to fall back to random crossover or mutation as it was often the case in [5]. For this purpose, a tree-based grammar guided genetic programming system (GGGP) [4] is used that was developed for program synthesis problems. It uses multiple grammars, one grammar for each available data type, and automatically combines them depending on the data types required for a problem at hand. The number of variables per type available to the program and a maximum execution time have to be set beforehand. The parameter settings have been taken from [4] and are summarized in Table 1.

A set of problems from the general program synthesis benchmark suite [8] is used, namely Checksum, Compare String Lengths, Double Letters, Grade, Mirror Image, Small Or Large, Sum of Squares and Vector Average, which are of varying difficulty and require various data types. Lexicase selection [9] is used as selection operator, as it has proven to be effective for program synthesis problems. In contrast to other selection operators that rely on a single fitness value for selection, lexicase uses the fitness values of every single training case. Individuals are selected based on the fitness achieve on randomly selected training case. In case of a tie, a subset of individuals that achieved the best result on the selected training case are taken and another training case is selected for comparison. This selection strategy gives overall poor performing individuals that may only solve a few or even a single training case perfectly a chance to be in the next generation.

Max_Tries is the only additional parameter that specifies how many times a semantic operator is allowed to try to select or generate a new subtree and do a semantic comparison. A similar parameter was also required for SCPS and MSSC.

The experiments are carried out with the semantic crossover and mutation operators introduced in this study, ESCPS and ESMPS, and for comparison with the conventional subtree crossover and subtree mutation [16].

The experiments have been executed with HeuristicLab [18]. An implementation of ESCPS and ESMPS, the benchmark suite [8] as well as lexicase selection [9] is available online [3].

5 RESULTS

This section discusses the results of the experiments carried out with the effective semantic operators and conventional subtree operators. The overall success rates, as well as semantic aspects of the operators, are discussed. The following results are mainly focused on crossover since it is the operator that is mainly used and therefore of higher interest, the plots for mutation are similar and that not enough space is available to show the plots for crossover and mutation.

5.1 Successful Runs and Fitness

The overall number of successful runs, the average test fitness of the best individuals, the average percentage of training and test cases solved for the semantic operators are shown in Table 2 as well as the improvements compared to conventional subtree operators. Additionally, a Wilcoxon rank sum test on the test fitness of the best training individuals was carried out to check for statistically significance. Four out of eight problems show a statistically significant difference in the results obtained with standard subtree operators. This has not been achieved for even one problem with SCPS.

The table shows that in almost all cases the semantic operators have improved the results on the benchmark problems. Sum of Squares gained the most successful runs due to the ESCPS and ESMPS. Some other problems suffer from overfitting, which was also reported in [8] and [4], but the increase of successful solutions found on training and improvements on average test fitness indicate that the semantic operators are beneficial for those problems. The percentage of solved test cases has not improved on all problems, but the amount it has decreased by is negligible, compared to the improvements that were achieved in most cases.

In Figure 2, notched box plots of the test fitness of the best individual are shown. The fitness of the runs with the semantic operators (Semantic) are compared with the conventional subtree operators (Normal). In four cases, Compare String Lengths, Double Letters, Sum of Squares and Vector Average, the results with the semantic operators are significantly better than with subtree operators. For Double Letters and Vector Average, it is slightly difficult to see due to scale for the outliers. For the other four problems, the fitness is similar. The semantic operators were able to significantly improve on 50% of the problems tackled and had little effect on the other. So, in the worst-case scenario, some computational overhead is wasted with ESCPS and ESMPS, but the results do not become worse.

5.2 Semantics

One of the shortcomings of SCPS was that random crossover was used up to 50% of the time and ESCPS tries to address this problem. Figure 3 shows the semantic measure used for comparison. 'Partial change' is the default semantic measure used, as explained in Section 3.1. If no subtree with a partial change is found, 'Any change' will be accepted. As a last resort, random crossover will be used. It should be noted that it is possible, that no crossover happens if for Max_Tries no subtrees of the same type can be found, as grammar guided GP is strongly typed, but this only happens in rare cases. The graphs for Compare String Lengths, Double Letters, Grade and Mirror Image are omitted, as they are very similar to Small Or Large and do not add any additional insight.

For all problems 'Partial change' is used most of the time, in many cases close to 100% of the time. This shows that this semantic measure is able to use the semantic information of subtrees more often that in case of SCPS in [5]. The second semantic measure 'Any change' is rarely used and only during the initial generations of GP, because 'Any change' is solely used if 'Partial change' fails and 'Any change' is a slightly more general measure than 'Partial change'. Even with this additional semantic measure, random crossover might be used more often, during the first few generations, but declines quickly.

5.3 Semantic Comparisons

Semantic comparisons are computationally expensive and a drawback of semantic operators. Figure 4 shows the average number of semantic comparisons that were required until a pair of subtrees were selected for crossover for all problems. If the second semantic 'Any change' was used, the number of comparisons was already at the maximum of 10.

Section 5.2 showed that in the initial generations it is more difficult to find a pair of subtrees with semantic differences, which explains why initially the number of comparisons is high in Figure 4, but it quickly declines for all problems and stabilizes around 2 to 3. This shows that the number of comparisons is not even close to the maximum except in the initial generations and that the computational overhead of the semantic operator is on average low and less than for SCPS.

5.4 Parent Comparison

When the semantic operator is used and does not fall back to the conventional operator, the subtrees that are exchanged have a different semantics. But this does not mean that the overall semantics of the whole individual changes. Figure 5 depicts the percentage of children that have semantics different to their rooted parent for all problems. The rooted parent is the one a subtree is removed from and the child shares the same root node with.

The percentage of children that are semantically different from their rooted parent is high but lower than the percentage of times semantic crossover is used, as can be expected, because not every semantic operation on a subtree automatically leads to a change in the overall semantics. In case of Compare String Lengths, Grade and Small Or Large, the percentage declines. At this point, no explanation was found for this behaviour. It might be the case that

Table 2: Results on the benchmark problems over 100 runs. The table shows the absolute number of successful runs on test and training, the average test fitness of the best training individual, the average percentage of training and test cases solved for the experiments with the semantic operators. The results are compared with standard subtree operators. A Wilcoxon rank sum test was carried out to check for statistical significance and the p-value is reported.

Problem Name	Test	Training	Avg Fitness (% Improv.)	Solved Training Cases	Solved Test Cases	p-value
Checksum	0 (+0)	0 (+0)	19309.74 (-0.74%)	11.78% (+1.21%)	2.83% (-0.04%)	0.1842
Compare String Lengths	4 (+2)	100 (+3)	86.64 (+35.25%)	100.00% (+0.03%)	91.34% (+3.05%)	**0.0310**
Double Letters	0 (+0)	0 (+0)	4380.81 (+2.85%)	25.87% (+2.58%)	12.32% (+0.93%)	**0.0076**
Grade	28 (-3)	81 (+0)	177.82 (+38.54%)	98.66% (+1.42%)	96.54% (+1.13%)	0.3276
Mirror Image	0 (+0)	70 (+19)	343.29 (-1.75%)	99.48% (+0.55%)	65.67% (-0.60%)	0.6521
Small Or Large	5 (-2)	66 (+15)	561.77 (+15.88%)	98.46% (+2.18%)	88.60% (+ 2.14%)	0.3332
Sum of Squares	13 (+10)	14 (+11)	184463.01 (+42.03%)	29.76% (+18.96%)	26.72% (+17.86%)	**0.0003**
Vector Average	5 (+0)	5 (+0)	95016.59 (+4.19%)	6.95% (+0.23%)	5.40% (-0.12%)	**0.0203**

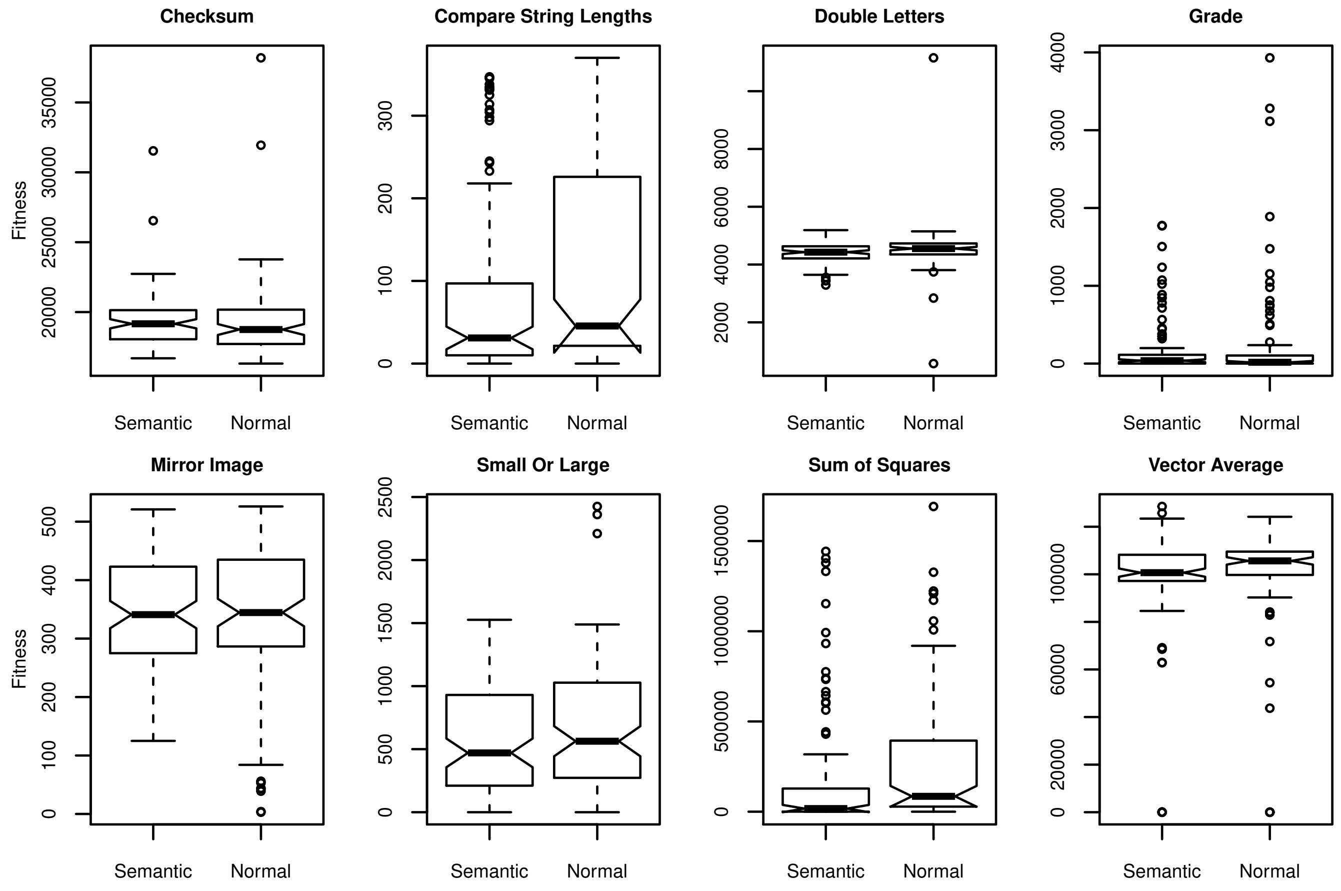


Figure 2: Notched box plots of the test fitness of the best individual during training comparing the semantic operators (Semantic) to the syntactical subtree operators (Normal).

the solutions become more robust to changes over generations for these problems.

Changing the semantics of an individual is important to keep semantic diversity high but does not automatically lead to better solutions. An additional experiment was carried out that checked the performance of the produced children during crossover for semantic and subtree crossover. Figure 6 shows the percentages of children that are better than their rooted parent and both parents for crossover. For all problems, ESCPS achieves a higher percentage of children that are better than their parents than subtree crossover.

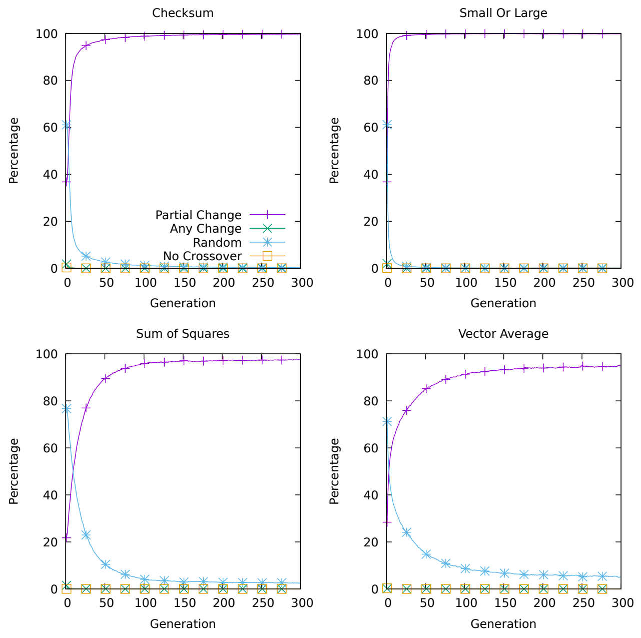


Figure 3: Percentage of semantic used during crossover over generations. 'Partial change' is the semantic that is used first. If no subtree pair for crossover is found 'Any change' is used, before falling back to 'Random' crossover. 'No crossover' is has taken place. The graphs for Compare String Lengths, Double Letters, Grade and Mirror Image are omitted, as they are very similar to Small Or Large and do not add any additional insight.

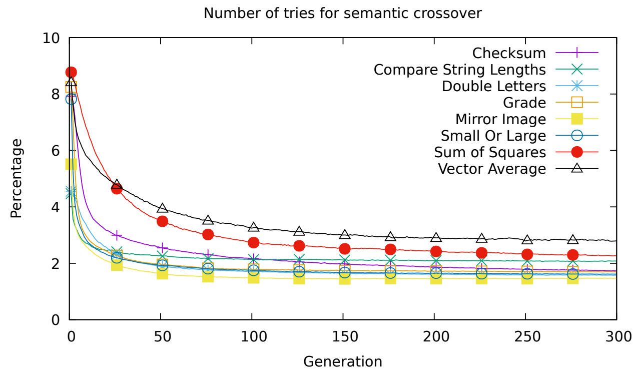


Figure 4: Number of tries subtrees selected for semantic comparisons until a subtree was selected or a maximum number was reached.

Similar trends have been found for mutation, but the plots are omitted due to space constraints. In case of Vector Average, even the percentage of children that are better than both parents with semantic crossover is close the percentage of children that are only better than the rooted parent with subtree crossover.

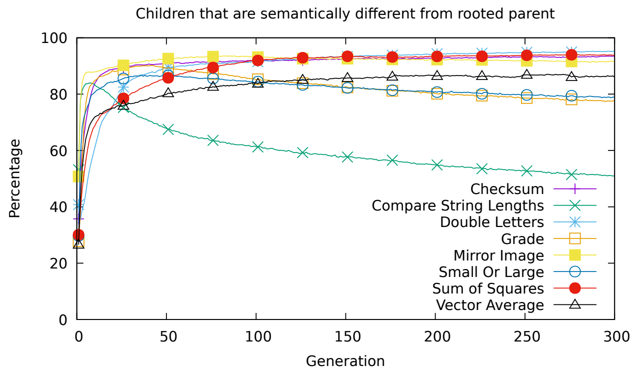


Figure 5: Percentage of children that produce a different semantics than their rooted parents during semantic crossover.

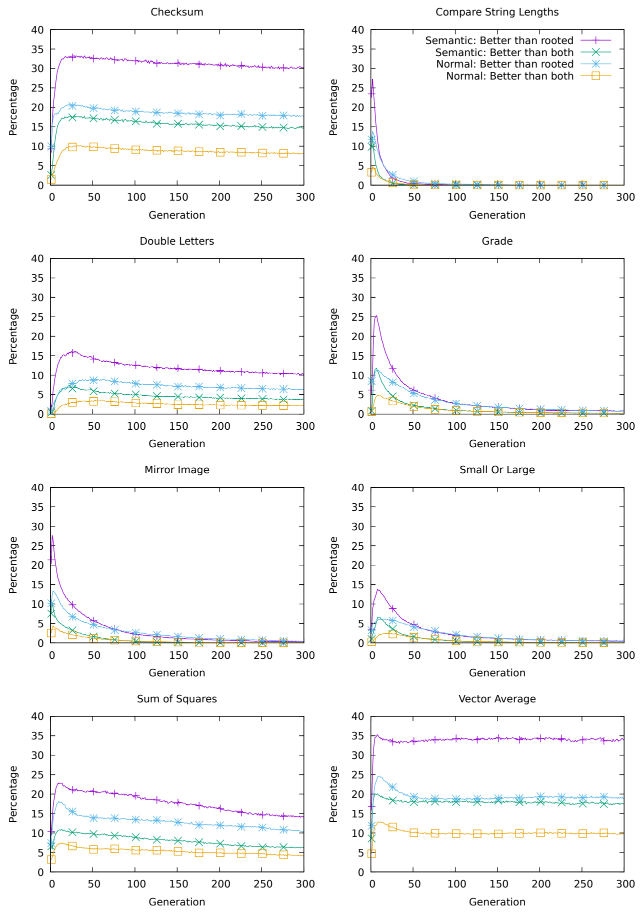


Figure 6: Percentage of children that are better than their rooted parent and both parents over generations during semantic crossover.

It is expected that lines decrease over time as runs will have solved the problem and continue for the purpose of this analysis, even though the run will not be able to create better individuals. This is even the case when a solution only solves all training but not test cases. An extreme case is Compare String Lengths, where all runs have been able to solve the training data. The percentage of children that are better than their parents rapidly decreases and gets close to zero. In that case, even subtree crossover achieves a higher percentage than semantic crossover at around generation 25, but only because more runs with ESCPS have already been solved, at least in training. A similar effect can be seen for Grade, Mirror Image and Small Or Large.

6 CONCLUSION & FUTURE WORK

In this study novel and effective semantic operators for program synthesis, ESCPS and ESMPS, have been introduced. These operators are adaptations of the semantic crossover for program synthesis [5] and improve it by addressing its shortcomings. ESCPS and ESMPS are able to effectively use the semantic information available almost all of the time in contrary to SCPS by using a simpler semantic measure and selecting pairs of subtrees instead of comparing a single subtree to multiple others. These effective semantic operators were able to produce more children that were improvements over their parents as well as achieve statistically significantly better results than conventional subtree operators. The latter was not achieved with SCPS.

Future work for semantic operators for program synthesis includes a more extensive study on the full set of benchmarks from the general program synthesis benchmark suite. The purpose of this paper was to show that effective semantic operators for program synthesis can be created, but it would be helpful to know on which kinds of problems it is more likely to improve performance with semantic information. Additionally, it should be tested, if it is possible to create a semantic measure that can estimate the similarity between two subtrees more precisely than the partial change or any change measure used in this paper and still be effective, as semantic locality is of importance to improve performance.

7 ACKNOWLEDGMENTS

This research is based upon works supported by the Science Foundation Ireland, under Grant No. 13/IA/1850.

REFERENCES

[1] Lawrence Beadle and Colin Johnson. 2008. Semantically Driven Crossover in Genetic Programming. In *Proceedings of the IEEE World Congress on Computational Intelligence*, Jun Wang (Ed.). IEEE Computational Intelligence Society, IEEE Press, Hong Kong, 111–116. https://doi.org/doi:10.1109/CEC.2008.4630784

[2] L. Beadle and C.G. Johnson. 2009. Semantically driven mutation in genetic programming. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on.* 1336–1342. https://doi.org/10.1109/CEC.2009.4983099

[3] Stefan Forstenlechner. 2016. Github repository: HeuristicLab.CFGGP: Provides Context Free Grammar Problems for HeuristicLab. (2016). https://github.com/t-h-e/HeuristicLab.CFGGP [Online; accessed 14-November-2016].

[4] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. *A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming.* Springer International Publishing, Cham, 262–277. https://doi.org/10.1007/978-3-319-55696-3_17

[5] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. Semantics-based Crossover for Program Synthesis in Genetic Programming. In *Artificial Evolution*, Evelyne Lutton, Pierrick Legrand, Pierre Parrend, Nicolas Monmarché, and Marc Schoenauer (Eds.). Springer International Publishing, Cham. https://ea2017.inria.fr//EA2017_Proceedings_web_ISBN_978-2-9539267-7-4.pdf

[6] Stefan Forstenlechner, Miguel Nicolau, David Fagan, and Michael O'Neill. 2015. Introducing Semantic-Clustering Selection in Grammatical Evolution. In *GECCO 2015 Semantic Methods in Genetic Programming (SMGP'15) Workshop*, Colin Johnson, Krzysztof Krawiec, Alberto Moraglio, and Michael O'Neill (Eds.). ACM, Madrid, Spain, 1277–1284. https://doi.org/doi:10.1145/2739482.2768502

[7] E. Galván-López, B. Cody-Kenny, L. Trujillo, and A. Kattan. 2013. Using semantics in the selection mechanism in Genetic Programming: A simple method for promoting semantic diversity. In *Evolutionary Computation (CEC), 2013 IEEE Congress on.* 2972–2979. https://doi.org/10.1109/CEC.2013.6557931

[8] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference.* ACM, Madrid, Spain, 1039–1046. https://doi.org/doi:10.1145/2739480.2754769

[9] T. Helmuth, L. Spector, and J. Matheson. 2015. Solving Uncompromising Problems With Lexicase Selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (Oct 2015), 630–643. https://doi.org/10.1109/TEVC.2014.2362729

[10] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. 2008. *Semantic Building Blocks in Genetic Programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 134–145. https://doi.org/10.1007/978-3-540-78671-9_12

[11] Alberto Moraglio, Krzysztof Krawiec, and ColinG. Johnson. 2012. Geometric Semantic Genetic Programming. In *Parallel Problem Solving from Nature - PPSN XII*, CarlosA.Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone (Eds.). Lecture Notes in Computer Science, Vol. 7491. Springer Berlin Heidelberg, 21–31. https://doi.org/10.1007/978-3-642-32937-1_3

[12] Quang Uy Nguyen, Xuan Hoai Nguyen, and Michael O'Neill. 2009. Semantic Aware Crossover for Genetic Programming: The Case for Real-Valued Function Regression. In *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009 (LNCS)*, Leonardo Vanneschi, Steven Gustafson, Alberto Moraglio, Ivanoe De Falco, and Marc Ebner (Eds.), Vol. 5481. Springer, Tuebingen, 292–302. https://doi.org/doi:10.1007/978-3-642-01181-8_25

[13] Quang Uy Nguyen, Xuan Hoai Nguyen, and Michael O'Neill. 2009. Semantics based Mutation in Genetic Programming: The case for Real-valued Symbolic Regression. In *15th International Conference on Soft Computing, Mendel'09*, R. Matousek and L. Nolle (Eds.). Brno, Czech Republic, 73–91. http://ncra.ucd.ie/papers/mendel2009SSM.pdf

[14] Quang Uy Nguyen, Xuan Hoai Nguyen, Michael O'Neill, R. I. McKay, and Edgar Galvan-Lopez. 2011. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines* 12, 2 (June 2011), 91–119. https://doi.org/doi:10.1007/s10710-010-9121-2

[15] Quang Uy Nguyen, Xuan Hoai Nguyen, Michael O'Neill, R. I. McKay, and Dao Ngoc Phong. 2013. On the roles of semantic locality of crossover in genetic programming. *Information Sciences* 235 (20 June 2013), 195–213. https://doi.org/doi:10.1016/j.ins.2013.02.008

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

[17] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. 2014. A survey of semantic methods in genetic programming. *Genetic Programming and Evolvable Machines* 15, 2 (2014), 195–214. https://doi.org/10.1007/s10710-013-9210-0

[18] Stefan Wagner, Gabriel Kronberger, Andreas Beham, Michael Kommenda, Andreas Scheibenpflug, Erik Pitzer, Stefan Vonolfen, Monika Kofler, Stephan Winkler, Viktoria Dorfer, and Michael Affenzeller. 2014. *Advanced Methods and Applications in Computational Intelligence.* Topics in Intelligent Engineering and Informatics, Vol. 6. Springer, Chapter Architecture and Design of the HeuristicLab Optimization Environment, 197–261. https://doi.org/10.1007/978-3-319-01436-4_10