Preprint version of: Saini, Anil Kumar, and Lee Spector. "Using modularity metrics as design features to guide evolution in genetic programming." *Genetic Programming Theory and Practice XVII.* Springer, Cham, 2020. 165-180.

Using Modularity Metrics as Design Features to Guide Evolution in Genetic Programming

Anil Kumar Saini, Lee Spector

Abstract

Genetic Programming has advanced the state of the art in the field of software synthesis. However, it has still not been able to produce some of the more complex programs routinely written by humans. One of the heuristics human programmers use to build complex software is the organization of code into reusable modules. Ever since the introduction of the concept of Automatically Defined Functions (ADFs) by John Koza in the 1990s, the genetic programming community has expressed the need to evolve modular programs, but despite this interest and several subsequent innovations, the goal of evolving large-scale software built on reusable modules has not yet been achieved. In this chapter, we first discuss two modularity metrics—reuse and repetition—and describe the procedure for calculating them from program code and corresponding execution traces. We then introduce the concept of design features, which can be used in addition to error measures to guide evolution, and demonstrate the use of modularity design features in parent selection.

1 Introduction

Modularity is ubiquitous in nature. Most of the systems around us — from our brains to almost every software program ever written — are modular. In biological systems, modularity contributes to the evolvability of an organism, which is its ability to

Anil Kumar Saini
College of Information and Computer Sciences, University of Massachusetts, Amherst, MA, USA
e-mail: aks@cs.umass.edu

Lee Spector
Department of Computer Science, Amherst College, Amherst, MA, USA, and School of Cognitive Science, Hampshire College, Amherst, MA, USA, and College of Information and Computer Sciences, University of Massachusetts, Amherst, MA, USA e-mail: lspector@hampshire.edu

adapt to new surroundings over evolutionary time [1]. In software engineering, it is advantageous in more obvious ways; parts of the program can be changed without affecting the whole program, and modules can act as building blocks that can be arranged in different orders or added on top of other modules to form entirely new programs.

Although genetic programming can easily solve simple problems like the ones found in introductory programming textbooks [2], evolving significantly more complex programs that are nonetheless routinely written by humans is still well beyond the state of the art. How do humans do it? Among other heuristics, humans frequently use modularity to write complex programs; they organize their code into units that can be reused again and again. Since modularity is pervasive in almost all complex systems, it might be the ingredient that genetic programming needs to evolve complex software.

In this chapter, we describe two modularity metrics that were inspired by similar metrics in software engineering — reuse and repetition — and we show how these metrics can be used to define design features that influence parent selection, incentivizing the evolution of modular programs. Our experiments demonstrate that the use of this technique can significantly decrease the average size of successful programs.

In the following sections, we briefly present the history of prior work on modularity in genetic programming. We then present our modularity metrics and the methods by which they can be calculated. This is followed by discussions of how the metrics can be used during evolution, and of the automatic program simplification methods that play a role in that usage. We then present our experiments and their results, and finish with a summary of our conclusions and suggestions for future work.

2 Modularity in Genetic Programming

The concept of modularity has long been discussed in genetic programming. John Koza introduced the concepts of Automatically Defined Functions (ADFs) [3] and Architecture Altering Operations [4] in tree-based genetic programming that can be used to evolve reusable modules in the evolving programs. Many other researchers [12] used these concepts to build much more flexible ways of inducing modularity. Module Acquisition [5], for example, designates a group of nodes in a tree as a module and protects it from manipulation by genetic operators. Automatically Defined Macros (ADMs) [6] allows for the evolution of macros capable of altering program architectures. Hierarchical Locally Defined Modules (HLDM) [7] introduces a hierarchical way of defining and using modules in genetic programming. These and other related methods basically provide function-like templates to the evolving programs to modify and use during the course of evolution.

Other efforts include designing the genetic programming systems in such a way that modules can be defined and used within the evolvable code itself, instead of

being constructed from the templates provided prior to the start of evolution. For example, in PushGP [13], "code" itself is type, which can be manipulated using specific instructions. SignalGP [17] is another technique where the program is a collection of modules. The concept of tags [14], which provides a procedure-calling mechanism during evolution, has also been introduced to encourage the adoption of modules.

Although the importance of modules has long been felt and efforts have been made to help programs acquire modules during the process of evolution, a very few attempts have been made to actually measure modularity. One such attempt is Functional Modularity [18], which considers modules as functional units and calculates the modularity based on their performance on a set of test cases. There are certain other measures of modularity [19, 20], which can only be used for programs that can be represented as networks.

Despite these efforts, evolving modularity is still termed as one of the open issues in genetic programming [11]. Most of the above-mentioned strategies to evolve modular programs try to give evolution access to the tools it may need to develop modules without offering any real incentive to actually use them. Additionally, these strategies make certain assumptions about the structure of the program, and consequently, some parameters like the number, types, or other information about modules need to be specified in advance before the evolution can start.

One of the motivations to push genetic programming systems to produce modular programs is that evolution often prefers non-modular programs. To understand this, let us look at a hypothetical scenario. Imagine there are two programs with the same error vectors solving the same problem: one with a module reused multiple times, and another without any module. During mutation in the first program, if any change occurs to this module, we would notice a huge change in the error vector, but the same might not happen in the case of the non-modular program.

Although modularity as a concept is not new to the field of software engineering, there have not been many attempts at defining modularity in the context of automatic programming, where instead of humans, machines write the programs. This chapter discusses the modularity metrics introduced in [9] and proposes the ways in which they can be used during evolution to incentivize the development of modules.

3 Modularity Metrics

Depending on the field of study, there can be different types of modularity. For example, [10] mentions developmental, morphological, evolutionary, and other kinds of modularity. Also, there are corresponding metrics to measure them. The Q-metric [19, 20], for example, is used to calculate modularity in networks.

In this chapter, we focus on the concept of modularity used in software engineering. Programmers often employ modules to solve parts of a given problem and reuse them multiple times to avoid code repetition. They also use many metrics to calculate the modularity of a given software. Coupling, for example, measures the

interdependence between different modules; cohesion, on the other hand, measures the amount of interaction among components of a particular module [8]. A modular software design, therefore, is characterized by low coupling and high cohesion.

General-purpose programming languages like Python, Java, etc. contain constructs in the form of functions, classes, etc. that can be used to identify a module. Genetic programming systems, on the other hand, do not have such standard constructs and often use different ways of achieving modularity. Due to this, it becomes difficult to detect modules in different representations in genetic programming, and the metrics used in software engineering might not work for evolved programs.

Nevertheless, the metrics presented in this chapter are inspired by some of the concepts used in software engineering like code reusability, component-based development, etc. Reuse and Repetition — two metrics presented in this chapter — are inspired by heuristics humans use to write programs; in order to have less repetition, they try to reuse their code using features like functions, procedures, etc.

Since we want a formulation that could be used irrespective of the underlying genetic programming system, we use the information contained in the execution trace of a program to calculate the metrics. Moreover, we maintain that instead of one single metric, there can be multiple metrics measuring different aspects of modularity.

In this section, we first define a module in the context of automatic programming. And then after going over the design principles used to formulate the metrics, we provide the exact equations to calculate those metrics.

3.1 Module

A module may be defined as a part of the program, which can be used and modified independently of other parts [9]. For the purpose of this chapter, a collection of tokens - keywords, variables, constants, etc. can be considered as a module. For a group of tokens to become a module, however, the following conditions must be met:

1. The order in which the tokens appear in a module should be the same as the order in which they appear in the program. For example, if we have a set of instructions `ABC` in the execution trace, for it to be considered a module, the same set of instructions should be present in the program in the same order.
2. The module should have a definite beginning and an end. For a group of instructions, as in most of the programming languages including Push, this is indicated by brackets, keywords, or indentations. In the case of single instructions, however, no such construct is needed.

3.2 Design Principles for Modularity Metrics

In order to come up with exact formulations for Reuse and Repetition, some design principles were used. The formulations which were in direct conflict with any of these principles were rejected, and those which complied with all of them were taken up for consideration. And the final set of metrics we present in Section 3.4 is one of them. The design principles are:

1. The frequency with which a module gets executed should contribute more than its size towards the metrics.
2. All consecutive sub-sequences of a module are also considered as modules. For example, if `ABC` is a module, `A`, `B`, `C`, `AB`, and `BC` are also modules.
3. Since we are extracting modules from execution traces, there might be some instructions in the trace which were not there in the program. This can happen when an instruction from the program calls instructions from other programs or libraries. To keep things simple, we will not use such instructions in our calculation of the metrics.
4. The modularity metrics measure the structural properties of a program. Therefore, the formulations of these metrics should not take into account the usefulness or other functional aspects of modules.
5. The modules should appear or be used at least twice for them to be considered in the calculation. This helps in identifying the boundaries of modules.
6. Since both Reuse and Repetition considers the frequency and size of modules, and they measure similar properties of modules, they can have similar formulations. They should, however, be independent of each other. In other words, a program can have a high value of Reuse with a low value of Repetition and vice-versa.

3.3 Reuse and Repetition

Reuse measures how frequently a copy of a module gets executed. Repetition, on the other hand, measures how frequently a given module appears in the program. The main difference between the two is that in the former one, one copy gets executed multiple times, whereas, in the latter, there are already multiple copies present in the program, each of which gets executed once.

A program will have high reuse if it has a function that is called repeatedly or a block of code that gets iterated multiple times. The program will have a high value of repetition if the same set of instructions are written multiple times and executed separately.

3.4 Reuse and Repetition from Execution Trace

An execution trace is the sequence of instructions arranged by the order in which they are executed. This sequence is often different from the actual program since the instructions at a particular position (say, a line number) in the program can call instructions at other locations.

As previously described in [9], the procedure to calculate the modularity metrics from the execution trace is as follows.

1. Assign an identifier to every token in the program. For simplicity, let this identifier be the position of the token in the program.
2. Execute the program. Maintain two traces, one with instructions (called the execution trace) and the other with identifiers (called the metadata trace). During the course of execution, if we come across an instruction that was not present in the program, we give it a special identifier. Such instructions will not be considered while computing the metrics.

Usually, in order to calculate error vector of a given program, it is executed on a set of test cases. Consequently, we have multiple executions traces for a single program. And since the metrics are calculated on the execution traces, we will have a list of values instead of a single number for each metric. To simplify this, we can use one of the two approaches: use summary statistics (for example, mean, max, etc.), or choose a value randomly from the list. As mentioned above, we will have two types of traces after the program gets executed. From these traces, modules can be extracted in many ways. For the sake of simplicity, a module is any group of instructions or identifiers that is repeated at least twice in the respective trace.

Now, a simple and naive way to quantify the amount of reuse and repetition is to compute the proportion of execution trace under reuse or repetition. The corresponding formulation for Reuse (U) and Repetition (P) would be:

$$U = \frac{\sum_{i=1}^{m} l_i \cdot f_i}{l} \tag{1}$$

and

$$P = \frac{\sum_{i=1}^{n} l_i \cdot f_i}{l}, \tag{2}$$

where there are m modules being reused, n modules being repeated, l_i and f_i are respectively the length and size of a module i, and l is the length of the trace. The length of the trace is simply the number of instructions present in it. Moreover, both traces have the same length. The main issue with this formulation is that they give equal weight to the size and frequency of the module, which is in direct contradiction to the design principles given in Section 3.2.

After increasing the weight of the frequency and normalizing it accordingly, another possible formulation can be:

$$U = \frac{\sum_{i=1}^{m} l_i \cdot 2^{f_i}}{2^l} \tag{3}$$

and

$$P = \frac{\sum_{i=1}^{n} l_i \cdot 2^{f_i}}{2^l}. \tag{4}$$

This formulation is also problematic. Since the measures give more importance to the frequency of usage (exponentiation) than to the length of a module (simple multiplication), we get higher value of Reuse when we have more frequent modules in an execution trace of a given length. In other words, as the length of the trace increases, the reuse measure starts to prefer short and frequent modules over bigger and less frequent ones. Table 1 illustrates this point. Although intuitively reuse should increase since the module `ABC` is getting reused more often, the actual reuse calculated from Equation 3 is decreasing. Similar issues occur during Repetition calculations.

Table 1 Reuse values of some toy examples calculated using Equation 3. Each letter denotes an instruction in the execution trace.

Execution Trace	Reuse Value
`ABCABCDEF`	0.078125
`ABCABCABCDEF`	0.01953125
`ABCABCABCABCDEF`	0.0048828125
`ABCABCABCABCABCDEF`	0.001220703125

The final formulations change the denominator to remedy this. The following equations are simplified versions of the equations given in [9]:

$$U = \frac{\sum_{i=1}^{m} l_i \cdot 2^{f_i}}{2^u} \tag{5}$$

and

$$P = \frac{\sum_{i=1}^{n} l_i \cdot 2^{f_i}}{2^v}. \tag{6}$$

where u is the number of unique identifiers used in the metadata trace, and v is the total number of instructions of the program used in the execution trace. These numbers can be different from the length of the program as some instructions get executed multiple times, and some might not get executed at all due to conditional operations.

While we presented general formulations of the metrics in this section, the exact procedure to extract modules from the evolving programs depends on the language in which they are represented.

4 Using Modularity Metrics to Guide Evolution

Every program generated by genetic programming has certain 'Software Quality Features.' One of these features is correctness (errors on test cases). Although correctness - which focuses only on the 'output' of a program, not its 'structure' - might be the most important feature, it is not the only one. There are many other features that focus on the structure of the generated programs. In this chapter, we will term these features 'design features' and differentiate them from 'correctness features.' We will use error on test cases as correctness feature and modularity metrics as design features.

In this section, we will explore ways to combine modularity metrics with errors of individuals to guide the evolution of programs.

4.1 Using Design Features during Parent Selection

The procedure to combine design features with error values will be different for different parent selection methods. Here, we describe the procedure with lexicase selection [15]. What we essentially aim to do is to have additional pressure to prefer more modular programs during parent selection.

In each iteration of Lexicase selection, the test cases (or, the error values on test cases) are shuffled in random order. The design features can be sorted among these error values in one of the following ways:

1. The first option is to shuffle error values and design features together.
2. The second option is to consider design features after error values. What this means is that out of two individuals, one with higher values of design features would be preferred over the other only if they both have the same error values.
3. The third option is to consider design features before error values. What this means is that out of two individuals, one with higher values of design features would always be preferred over the other, irrespective of error values.

We note that modularity metrics are used during selection just like normal error values, but they are not taken into consideration when determining whether a program is a solution to a problem or not.

Another method of using modularity metrics to guide the evolution of modular programs is to filter out individuals with low values of modularity metrics from the population before the parent selection commences.

4.2 Using Design Features during Variation

Modularity metrics can also be used to guide the mutation and crossover operators. For example, the values of reuse and repetition can serve as inputs to a given

variation operator, and the rate and other parameters of variation can be decided accordingly. Additionally, instead of applying a variation operator uniformly, we can use the reuse and repetition values of the parts of a program to decide the location of variation.

5 Experiments and Results

In this section, we describe the experiments that were conducted to investigate the effects of using modularity metrics as design features. We run our experiments on Clojush[1] (Clojure implementation of PushGP) which evolves programs in a stack-based programming language called Push. Since calculating modularity metrics is time-intensive, we decided to use the problem of symbolic regression because of its faster runtime per generation compared to other more complex problems. Specifically, we try to evolve programs to compute the mathematical expression of $(x^3+1)^3+1$.

Before presenting our experimental set-up, we first describe the procedure to extract from Push programs the modules to be used to compute the metrics. We also introduce the concept of autosimplification as a way to remove unnecessary instructions from the programs prior to calculating metrics.

5.1 Extracting Modules from Push Programs

As mentioned in Section 3.4, the procedure to retrieve modules from the execution trace of a given program depends on the language in which the program is represented. Since we are using PushGP in our experiments, we present the algorithm to extract modules from Push programs running on that system. The same procedure can be used for other systems with minimal changes.

Push is a stack-based programming language with separate stacks for every data type [13]. During execution, the instructions can take their inputs from and place their outputs on different stacks. In each iteration, the top element of the execution stack gets executed. Hence, the sequence of the top elements on the execution stack after every iteration becomes the execution trace.

Algorithm 1, which describes the procedure to calculate the metrics for Push programs, can also be used for programs written in similar stack-based programming languages. Table 2 gives an example of Push program and the corresponding execution and metadata traces.

We first covert the metadata trace into a different representation in the following way. A single instruction is represented as `[a:a]` where `a` is the identifier of that instruction. A group of instructions inside parentheses, with the identifier of the first

[1] https://github.com/lspector/Clojush

instruction being `a` and that of the last one being `b`, is represented as `[a:b]`. Additionally, modules in an execution trace are called instruction-modules (for example, `(exec_dup (exec_swap 1 2))`), and those in metadata trace are called identifier-modules (for example, `[1:4]`).

Table 2 A Push program example and the corresponding execution and metadata traces.

Program	`(exec_dup (exec_swap 1 2))`
Execution Trace	`((exec_dup (exec_swap 1 2)) exec_dup (exec_swap 1 2) exec_swap 2 1 (exec_swap 1 2) exec_swap 2 1)`
Metadata Trace	`((1 (2 3 4)) 1 (2 3 4) 2 4 3 (2 3 4) 2 4 3)`
Metadata Trace (modified)	`([1:4] [1:1] [2:4] [2:2] [4:4] [3:3] [2:4] [2:2] [4:4] [3:3])`

To extract modules for Reuse, we use the metadata trace. In each iteration of the outer loop, we search the metadata trace for modules of a certain size. Whenever we find a module of size s, from the next $(s+1)$ modules in the trace, we collect the ones which are 'included' in the bigger module. For example, if `[1:4]` is the module under consideration, the modules `[1:1]`, `[1:3]`, `[2:4]`, etc. are considered to be 'included' in the bigger module. Before deleting these smaller modules from the metadata trace, we use them to find modules containing consecutive identifiers as follows. **AllContinuousSeqs()** takes a set of identifier-modules, and output those proper subsets which contain consecutive identifiers. For example, if the input is $\{$`[2:2],[3:3],[4:4], [6:8]`$\}$, it will output $\{$`([2:2]), ([3:3]), ([4:4]), ([2:2], [3:3]), ([3:3],[4:4]), ([2:2],[3:3],[4:4]),([6:8])`$\}$. This is in tune with the design principles described in Section 3.2, which allows the subsets of modules to be considered as modules as well. Following this procedure, we will have a list of modules at the end of the outer loop. And the modules which appear at least twice in this list will we considered for computing Reuse.

Now, to get instruction-modules for Repetition calculation, we first get unique identifier-modules from the list described above, and look for the corresponding instruction-modules in the execution trace. From these modules, the ones which are repeated at least twice are considered for computing Repetition. Note that only the modules containing the same set of instructions with different identifiers are considered for calculating Repetition.

5.2 Autosimplification

Sometimes, the programs evolved by a given genetic programming system contain many instructions that do not contribute to its performance on test cases. In other words, some code elements can be removed from a program without affecting its performance on test cases. While these unnecessary instructions do not affect the overall error vector of the program, they can reduce the accuracy of modularity met-

Input: execution trace of the program containing instructions;
metadata trace of the program containing corresponding identifiers;
Result: Reuse and Repetition values
mTrace := modified metadata trace (see Table 2 for an example);
len := length of *mTrace*;
seqs := an empty set;
for $i = 1, 2, 3...len$ **do**
  for *module [m:n] in mTrace* **do**
    if $n - m == i$ **then**
      temp_items := take the next $(i+1)$ items $[m_j, n_j]$ from *mTrace* provided $[m_j, n_j] \neq [m, n]$ and $m_j \geq m$, and $n_j \leq n$;
      temp_items := sort the items by the first element of each pair;
      seqs := seqs + **AllContinuousSeqs**(temp_items);
      delete these items from *mTrace*;
    end
  end
end
seqs_for_reuse = seqs after removing the identifier-modules appearing only once;
use Equation 5 to calculate Reuse;
seqs_for_repetition = unique identifier-modules from seqs;
get the corresponding instruction-modules and their frequencies;
use Equation 6 to calculate Repetition;

Algorithm 1: Calculating metrics for Push programs

rics. Hence, we perform two sets of experiments, one where we calculate the metrics on non-simplified programs, and the other where we calculate them on simplified ones.

Algorithm 5.2 gives a simple procedure to automatically simplify a given Push program with certain number of steps. Although the procedure can produce different simplified programs each time it is run, the prior work shows that it often produces consistent results in practice. For programs written in other languages, similar algorithms can be developed.

5.3 Experimental Set-up and Results

As discussed in Section 4, there are multiple ways of using the modularity metrics to incentivize the development of modules in evolving programs. In our experiments, we focus on using the metrics during lexicase selection.

We follow the procedure laid down in Section 4.1. Prior to parent selection, we first compute the error on a set of test cases. Then to calculate the metrics, we choose a test case randomly from the list of test cases, execute the program on that test case, and then use the execution trace so obtained to compute the metrics. During lexicase selection, we shuffle the errors and metrics together for every selection event. In addition to individuals with low errors, we prefer the ones with high values of Reuse and low values of Repetition.

```
Input: the program to be simplified;
       number of steps;
Result: simplified program
repeat
    rand := a random number between 0 and 1;
    if rand < 0.5 then
        remove a small number (typically 1 or 2) of random instructions;
    else
        remove a random parenthesis pair;
    end
    calculate the error vector on new program;
    if error vector is same as before then
        update the program;
    else
        revert to the original;
    end
until the step limit is reached;
```

Algorithm 2: Auto-simplification

We perform four sets of experiments with 30 runs in each set: without using any metrics, using only reuse metric, using only repetition metric, and using both the metrics. In these experiments, the metrics are calculated on non-simplified programs. The parameters used during the runs are listed in Table 3. To deal with floating-point numbers, we use ε-lexicase [16] parent selection algorithm. We do not perform crossover and instead use UMAD (Uniform Mutation by Addition and Deletion)[21] mutation operator. During evolution, the programs have access to all the instructions that operate on float and execution stacks.

Table 3 Genetic Programming Parameters.

Parameter	Value
Population size	1000
Number of generations	500
Parent selection algorithm	ε-lexicase
Mutation operator	Uniform Mutation by Addition and Deletion
Mutation rate	0.09
Number of runs per condition	30

The results are given in Table 4. The number of successes is the number of runs out of 30 that evolved a successful program passing all the test cases. The size of a push program is the total number of instructions and parenthesis pairs present in it. In addition to presenting the average size of successful programs, we also show the average size of successful programs after simplification so as give a sense of the proportion of nonessential instructions in them. To test statistically significant differences, we used pairwise comparisons for proportions for success rates and the Mann-Whitney-Wilcoxon Test for the sizes of successful programs. From the table,

it is clear that using reuse as a design feature makes the successful programs smaller, without significantly affecting the success rate. However, when we use repetition metric either separately or with reuse, we get larger programs without any significant difference in success rates. This might be due to the fact that, in order to have a low value of repetition, the programs can often have useless instructions, which basically makes the execution traces bigger and hence increase the denominator in Equation 6. This is well supported by the last two rows of the table, where the difference between the sizes of programs before and after simplification is very large.

Table 4 Using metrics as design features. The metrics have been calculated on non-simplified programs. The underline indicates that the value is significantly smaller than the corresponding value in the no-intervention case.

	Number of successes	Average size of successful programs	Average size of successful programs after simplification
No intervention	12	43.08	20.42
Reuse only	12	<u>31.00</u>	14.67
Repetition only	5	60.80	26.8
Both metrics	7	58.71	11.86

To restrain the modularity metrics from preferring programs with more nonessential instructions, we used the concept of autosimplification as discussed in Section 5.2. In other words, prior to calculating the metrics, we simplified the program with the number of simplification steps being 50. Accordingly, we performed 30 runs for each of the conditions mentioned earlier. The results are given in Table 5. Although we do see a slight improvement in the success rate while using the reuse metric, the results are very similar to Table 4. We still get large programs while using the repetition metric as we did before. As a future work, we can either explore increasing the number of simplification steps or use some other mechanism to rein in the increase in the program size when using the repetition metric.

Table 5 Using metrics as design features. The metrics have been calculated on simplified programs, with the number of simplification steps being 50. The underline indicates that the value is significantly smaller than the corresponding value in the no-intervention case.

	Number of successes	Average size of successful programs	Average size of successful programs after simplification
No intervention	12	43.08	20.42
Reuse only	17	34.12	17.71
Repetition only	5	67.00	23.8
Both metrics	5	62.80	<u>8.6</u>

From the results described above, it can be concluded that the best results are obtained by computing the reuse metric on simplified programs and using it as a design feature in lexicase selection. However, these results are preliminary in nature

and are mainly intended to show how the modularity metrics can be used in the evolutionary framework.

6 Conclusions and Future Work

In this chapter, we first discussed a set of modularity metrics - reuse and repetition - that measure different aspects of modularity. We then presented multiple schemes of using these metrics in the framework of evolving programs. We demonstrated one of these ways by computing the metrics on Push programs and using them as design features in lexicase selection. We presented some preliminary results which show that using the reuse metric gives us more compact successful programs.

We use symbolic regression in our experiments, which is very easy to solve and might not even benefit from having modules. Therefore, as a future work, we need to run the same set of experiments on more complex problems like the ones from the benchmark suite of [2]. We can further optimize the whole procedure to calculate the metrics and also explore other methods of using them mentioned in Sections 4.1 and 4.2.

While the metrics presented in this chapter focus only on the structure of modules, new metrics that take into account the usefulness of those modules in solving different test cases can also be added to the metrics suite.

Considering almost all complex systems around us are modular in nature, we might benefit substantially from encouraging modularity in genetic programming systems if we want to synthesize programs as complex as the ones written by human programmers. We also believe the lessons learned from employing modularity metrics in the evolutionary framework can help us develop the tools to tackle some of the most difficult and unsolved problems in genetic programming. Further experiments to investigate the utility of modularity in the evolution of programs may also shed some light on the role of modularity in the evolution of biological systems.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 1617087. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Clune, J., Mouret, J. B., & Lipson, H. (2013). The evolutionary origins of modularity. Proceedings of the Royal Society b: Biological sciences, 280(1755), 20122863.
2. Helmuth, T., & Spector, L. (2015, July). General program synthesis benchmark suite. In Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (pp. 1039-1046). ACM.
3. Koza, J. R. (1992). Genetic programming: on the programming of computers by means of natural selection (Vol. 1). MIT press.

4. Koza, J. R. (1994). Architecture-altering operations for evolving the architecture of a multi-part program in genetic programming.
5. Angeline, P. J., & Pollack, J. (1993, February). Evolutionary module acquisition. In Proceedings of the second annual conference on evolutionary programming (pp. 154-163).
6. Spector, L. (1995). Evolving Control Structures with Automatically De ned Macros. In Submitted to the 1995 AAAI Fall Symposium on Genetic Programming.
7. Banzhaf, W., Banscherus, D., & Dittrich, P. (1999). Hierarchical genetic programming using local modules. Secretary of the SFB 531.
8. Dhama, H. (1995). Quantitative models of cohesion and coupling in software. Journal of Systems and Software, 29(1), 65-74.
9. Saini, A. K., & Spector, L. (2019). Modularity Metrics for Genetic Programming. In Genetic and Evolutionary Computation Conference Companion (GECCO 19 Companion), July 1317, 2019, Prague, Czech Republic. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3319619.3326908
10. Callebaut, W., Rasskin-Gutman, D., & Simon, H. A. (Eds.). (2005). Modularity: understanding the development and evolution of natural complex systems. MIT press.
11. ONeill, M., Vanneschi, L., Gustafson, S., & Banzhaf, W. (2010). Open issues in genetic programming. Genetic Programming and Evolvable Machines, 11(3-4), 339-363.
12. Gerules, G., Janikow, C. (2016, July). A survey of modularity in genetic programming. In 2016 IEEE Congress on Evolutionary Computation (CEC) (pp. 5034-5043). IEEE.
13. Lee Spector. 2001. Autoconstructive evolution: Push, pushGP, and pushpop. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO- 2001), Vol. 137.
14. Spector, L., Martin, B., Harrington, K., Helmuth, T. (2011, July). Tag-based modules in genetic programming. In Proceedings of the 13th annual conference on Genetic and evolutionary computation (pp. 1419-1426). ACM.
15. Helmuth, T., Spector, L., & Matheson, J. (2015). Solving uncompromising problems with lexicase selection. IEEE Transactions on Evolutionary Computation, 19(5), 630-643.
16. La Cava, W., Spector, L., Danai, K. (2016, July). Epsilon-lexicase selection for regression. In Proceedings of the Genetic and Evolutionary Computation Conference 2016 (pp. 741-748). ACM.
17. Lalejini, A., Ofria, C. (2018, July). Evolving event-driven programs with SignalGP. In Proceedings of the Genetic and Evolutionary Computation Conference (pp. 1135-1142). ACM
18. Krzysztof Krawiec and Bartosz Wieloch. 2009. Functional modularity for ge- netic programming. In Proceedings of the 11th Annual conference on Genetic and evolutionary computation. ACM, 9951002.
19. Newman, M. E. (2006). Modularity and community structure in networks. Proceedings of the national academy of sciences, 103(23), 8577-8582.
20. Qin, Z., McKay, R., & Gedeon, T. (2018). Why don't the modules dominate-Investigating the Structure of a Well-Known Modularity-Inducing Problem Domain. arXiv preprint arXiv:1807.05976.
21. Helmuth, T., McPhee, N. F., & Spector, L. (2018, July). Program synthesis using uniform mutation by addition and deletion. In Proceedings of the Genetic and Evolutionary Computation Conference (pp. 1127-1134). ACM.