Modularity Metrics for Genetic Programming

Anil Kumar Saini
University of Massachusetts
Amherst, MA
aks@cs.umass.edu

Lee Spector
Hampshire College
Amherst, MA
lspector@hampshire.edu

ABSTRACT

With improvements in selection methods and genetic operators, Genetic Programming (GP) has been able to solve many software synthesis problems. However, so far, the primary focus of GP has been on improving success rates (fraction of the runs that succeeds in finding a solution). Less attention has been paid to other important characteristics and quality measures of human-written programs. One such quality measure is modularity. Since the introduction of Automatically Defined Functions (ADFs) by John Koza, most of efforts involving modularity in GP have been directed towards pre-programming modularity into the GP system, rather than measuring it for evolved programs. Modularity has played a central role in evolutionary biology. To study its effects on the evolution of software, however, we need a quantitative formulation of modularity. In this paper, we present two platform-independent modularity metrics, namely, reuse and repetition, that make use of the information contained in the execution traces of the programs. We describe the process of calculating these metrics for any evolved program, using problems that have been solved with the PushGP system as examples. We also discuss some mechanisms for integrating these metrics into the evolution framework itself.

CCS CONCEPTS

• **Software and its engineering → Genetic programming**; *Abstraction, modeling and modularity*;

KEYWORDS

Genetic Programming, PushGP, Modularity, Reuse, Repetition

ACM Reference Format:
Anil Kumar Saini and Lee Spector. 2019. Modularity Metrics for Genetic Programming. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3319619.3326908

1 INTRODUCTION

Genetic Programming (GP) has made huge contributions to the field of software synthesis, to the extent that it can now solve many introductory-level computer science programming problems [4]. With improvements in selection methods and genetic operators, GP has been able to increase success rates on the problems that have already been solved and also find solution to new problems. So far, we have been focusing only on whether a given evolved program is able to solve the problem or not. This contrasts with the case of human-written programs, for which we care not only about whether a program works correctly, but also about how it looks. In other words, we care about quality in addition to the correctness of programs. One such quality measure is modularity, which is defined as the degree to which a system can be decomposed into separate but inter-connected components. Modularity not only makes a program easier to understand, it might also be essential to developing more complex programs.

Modularity has long been studied in evolutionary biology. Many biological entities - such as, animal brains, gene regulation, protein interactions, etc. - are modular in nature. Although there does not seem to be consensus on how it originated in the first place, it certainly has some advantages - it contributes to the evolvability (ability to adapt to new environments) of biological organisms [1]. Modular structure is also useful in human written software in a number of ways. First, it enables the programmer to make changes to one module without affecting other modules. Second, modules can act as building blocks, whereby new modules can be added any time and existing modules can be combined to make more complex software. And since we expect GP to evolve software in a similar way, we need to measure modularity alongside fitness of the evolvable programs.

2 MODULARITY IN GENETIC PROGRAMMING

The discussion about modularity and its advantages in Genetic Programming can be traced back to the concept of Automatically Defined Functions (ADFs), introduced by John Koza in his first book on Genetic Programming [7]. Although ADFs in their original form are constrained in how they can be used in an evolving program, the concept itself has led to the development of other more flexible forms of inducing modularity, like Automatically Defined Macros (ADM), Module Acquisition (MA), Adaptive Representation through Learning (ARL), etc. [3]. Other GP systems like PushGP [14] provide further flexibility; PushGP has a built-in mechanism for the evolution of modules through code self-manipulation [15].

All the above-mentioned mechanisms facilitate the creation of modules during the process of evolution, but none of them try to measure the amount of modularity in evolved programs. Some efforts in this direction include Functional Modularity [8], which considers modules as functional units and takes into account their performance on test cases. Our metrics, on the other hand, define modules in a general sense and focus on how frequently the modules are being used, and not on their functionality. In the future,

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326908

such metrics that focus more on the functional aspect of modules can be considered along with the metrics introduced in this paper. In some cases, where the program is essentially a network, modularity is defined in terms of how decomposable the network is into separable but connected components [6]. Our metrics can calculate modularity for any program and do not require the program to be network-like.

3 MODULES

A module can be defined in a number of ways. In software engineering, for example, a module is a part of solution exhibiting some sort of independence [8], and can be reused any number of times while writing that solution. In evolutionary biology, a module is considered as a semi-autonomous entity that can evolves and function relatively independently from other modules [2]. In network science, a given network is considered modular if it contains highly connected clusters of nodes that are sparsely connected to nodes in other clusters [1].

To define a module for our measures of modularity, we refer to [5] which defines a module as an encapsulated group of elements in the program that can be manipulated as a unit. The exact definition of an element, however, will depend on the particular system we are measuring modularity for. For example, in a C++ statement "int x=5;", we can have as elements, the whole statement, or the tokens like "int", "x", etc. By the definition given above, a group of elements such as a labeled or unlabeled procedure, a chunk of code that gets iterated multiple times, etc., is a module. From now on, we will be using elements and instructions interchangeably.

We, however, impose certain conditions on a set of instructions for it to be termed as a module:

(1) All the elements comprising the module must be together in the program in sequence. For example, if there is a sequence "ABC" in the program, the possible modules are: "A", "B", "C", "AB", "BC", and "ABC". Set of instructions like "AC", "CBA", etc. will not be considered as modules.
(2) There should be a specific beginning and an end to a module. For example, if "ABC" is a module, it should start with "A" and end with "C" every time it appears in the program, irrespective of its location.
(3) While executing, the whole module should execute as a group. Chunking or splitting of a module is not allowed.

4 MODULARITY METRICS

There exists a well-defined measure to calculate modularity in networks [9, 10]. This metric is very useful in the domains where the structure of the object is network like or can be converted into one [1, 2, 11]. There are however, many domains - most of the genetic programming systems, for example - where either the object under consideration itself can not be converted to a network, or the analysis becomes very complex when we try to do so. Our measures of modularity fills this gap, at least for the field of genetic programming.

Depending on how one uses the information about the number, type, and size of modules, there can be multiple measures of modularity, which may or may not be combined into a single measure. In this paper, we present two such measures: reuse and repetition. Reuse is defined as a measure of the number of times a particular copy of a module gets executed. Repetition, on the other hand, is a measure of the number of times different copies of a module get executed.

5 MODULARITY BASED ON EXECUTION TRACE

In this section, we present measures of modularity calculated from the information given in the execution trace of a program. We also provide the the procedure to calculate these measures.

5.1 Design Principles

We have used the following principles to guide the design of our measures:

(1) Frequency of use should matter much more the length of a particular module. For example, a module of length two repeated four times should have more reuse/repetition than a module of length four repeated two times.
(2) A part of a module is also a module provided that it satisfies the conditions given in Section 3. For example, if "ABC" is a module, "A", "B", "C", "AB", "BC" are also modules.
(3) Reuse and repetition should have similar formulations since both of them use size and frequency of modules in a similar way.
(4) Reuse and repetition should be independent of each other. This means that a program can have high reuse with low repetition and vice-versa.

5.2 Execution Trace

The order in which instructions in a given program are executed can be different from the order in which they appear in the program. To calculate modularity metrics for any programming language, the execution trace in that language should give the exact order of execution of instruction and a way to identify the instructions in the trace, using either the exact text of the instruction or a pointer to it (for example, line number).

5.3 Reuse and Repetition

To calculate the measures, we need to assign unique identifiers to all the instructions of the program. For the sake of simplicity, we assign as identifiers the position of instructions in the program. This means, the even two instruction are same but they appear on two different locations, they will be given two different identifiers. If during the execution, an instruction appears that was not present in the original program, it is also assigned an identifier that is different from the ones already used. The identifier serves as metadata for the instruction during its execution. When the program gets executed, we obtain an execution trace. Since the instructions were accompanied by identifiers throughout the execution, we can extract two types of sequences from the trace - one of instructions and another of identifiers.

From this execution trace, we can calculate the reuse (U) as

$$U = \frac{\sum_{i=1}^{l} \sum_{j=1}^{l-i+1} i \cdot 2^{n_j^i}}{2^u} \tag{1}$$

and repetition (P) as,

$$P = \frac{\sum_{i=1}^{l} \sum_{j=1}^{l-i+1} i \cdot 2^{m_j^i}}{2^v}. \quad (2)$$

The parameters used in the above equations are described below:

- u is the number of instructions with different identifiers used in the execution trace, or, in other words, number of unique identifiers in the execution trace.
- v is the number of instructions of the program used in the execution trace, irrespective of their identifiers.
- n_j^i is the number of times jth module of length i appears in the execution trace (one set of instructions in the program being used multiple times in the execution trace). To calculate this number, we can use the meta-data associated with each instruction.
- m_j^i is the number of times different copies of jth module of length i appears in the execution trace (multiple copies of the same set of instruction having different identifiers). To calculate this, we have to use both instructions and their identifiers appearing on the execution trace.

We are only considering those modules which are executed at least twice ($n_j^i > 1$ and $m_j^i > 1$). This will simplify our calculations because otherwise, we will not be able to know the boundaries of the modules just by looking at the execution trace.

Reuse can have a maximum value of 2^l, where l is the length of execution trace. This happens when there is a program of single instruction and that instruction repeats l times in the execution trace. Similarly, Repetition can also have a maximum value of 2^l, when one instruction is repeated l times in the program. Both of these measures can have minimum value of zero. In most of the programs in real world, there are many unique instructions, with a very small number of instruction repeating in the execution trace. Hence,the denominator in Equations 1 and 2 becomes very large and the reuse and repetition usually have very small values.

Reuse and repetition can also be combined to get a single measure of modularity as $M = f(U, P)$, where f is a function such that it increases with increasing U and decreases with increasing P. We will not combine the two measures and will let the user of these measures define f according to their needs.

5.4 Example

We will now consider a toy example and calculate reuse and repetition for it. Consider the following program (each letter denotes a single instruction and for the purpose of simplicity, we have not show the arguments): *ABCABD*. And according to the procedure described in Section 5, we assign identifiers to all the instructions: *{A:1, B:2, C:3, A:4, B:5, D:6}*. Note that the identifier associated with each instruction is basically its location in the program. Assume we get the following execution trace: *{A:1, B:2, C:3, A:1, B:2, C:3, A:4, B:5}*.

For calculating Reuse, we look at the metadata associated with the trace (12312345). And hence we have the following modules (appearing at least twice in the trace), after converting idenfiers back to instructions: A, B, C, AB, BC, ABC. Note that only five unique identifiers have been used in the execution trace. Accordingly, we can calculate Reuse and as:

$$U = \frac{1 \cdot (2^2 + 2^2 + 2^2) + 2 \cdot (2^2 + 2^2) + 3 \cdot (2^2)}{2^5} = 1.25$$

Similarly, for Repetition, we look at the instruction in the trace (*ABCABCAB*). Here, we have the following modules (appearing at least twice in the trace): A, B, AB. Note that ABC is not a module for this measure because it has the same meta-data, implying it is the same copy used twice. Considering only three unique instructions are used here, we can calculate the measure as:

$$P = \frac{1 \cdot (2^2 + 2^2) + 2 \cdot (2^2)}{2^3} = 2.0$$

6 CALCULATING MODULARITY FOR PUSH PROGRAMS

In this section, we will look at the typical values of reuse and repetition for the programs evolved in Push[12] as examples. Though our measures are independent of the programming language and GP system in which a given programs has been evolved, we use Push language because it has some in-built features that allow for the expressions of modules [15]. One such feature is that "code" itself is a type in Push that can be manipulated and executed as a unit. We believe the process we describe here can be used in other types of GP systems with very few changes.

Push is a stack based programming language with separate stacks for each of the data types. During execution, the instructions 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. The GP system designed to evolve programs in Push is known as PushGP [12].

Using the procedure defined in Section 5, we calculate the modularity metrics for some of the evolved Push programs in PushGP. The programs with the corresponding reuse and repetition values are given in Table 1. In the next paragraphs, we will try to explain the modularity values for the examples in Table 1.

Example 1. The first example shows a program evolved for the Double Letters problem given in benchmarking suite of [4] (Section 4, Problem 5). The program has non-zero reuse because of the presence of instructions like `exec_while` and `exec_dotimes`. `exec_while` keep on executing the code on the top of exec stack as long the top element of boolean stack has the value *true*. Similarly, `exec_do*times` executes the code on the top of execution stack a certain number of times. Both of these instruction perform looping, meaning a set of instructions get executed multiple times. Hence, a non-zero value of reuse.

Example 2. The second example is an evolved program for symbolic regression problem, where we try to evolve program for $x^3 - 2x^2 - x$. This program has zero reuse as there are no looping instructions. It has a high value of repetition because most of the instructions are repeated in the program itself.

7 EVOLUTION GUIDED BY MODULARITY

It is quite probable that there does not exist any correlation between modularity and fitness of a given program. This is because a program with very low fitness value can get a good modularity value on account of it having a lot of useless functions. Similarly,

Table 1: Reuse and Repetition values for some programs. The instructions in bold are the looping instruction that increase the value of Reuse. The underlined instructions, on the other hand, are repeated in the program and hence, increase the value of Repetition. Note that there are more looping and repeated instructions than those highlighted here.

Sr. No.	Program	Problem	Reuse	Repetition
1	(integer_stackdepth string_flush string_yankdup integer_lte boolean_invert_first_then_and char_empty exec_empty boolean_swap string_conjchar boolean_invert_first_then_and char_isdigit string_yank in1 char_isdigit char_dup string_stackdepth print_string string_dup_items string_substring exec_stackdepth boolean_empty exec_noop string_substring **exec_while** (boolean_invert_second_then_and string_shove char_empty string_conjchar exec_s () (string_dup_times integer_div exec_swap (**exec_do*times** (string_replacechar char_allfromstring integer_eq integer_min integer_dec char_isletter integer_mult string_indexofchar integer_flush) integer_fromchar exec_string_iterate () string_butlast char_frominteger) (integer_dup_items string_split **exec_dup** (char_yankdup string_occurrencesofchar) integer_inc string_concat) string_take) () string_pop) exec_y (char_eq))	Double Letters	5.96E-8	1.91E-6
2	(integer_div integer_sub integer_mult integer_div integer_mult integer_sub integer_sub 2 integer_sub integer_div in1 integer_mult in1 integer_mult integer_mult integer_mult integer_sub in1 integer_sub integer_sub)	Symbolic Regression	0.00E0	1.71E1

a program with a high fitness value can have a low modularity because every modular program can be written in a monolithic fashion.

Though fitness and modularity are not correlated, we might still need modularity to help us build more complex programs. And one of the ways to evolve modular solutions for a problem is to exert an indirect selection pressure on the population. In this scheme, modularity becomes a secondary selection criterion, i.e., out of two individuals, the one having higher modularity value will be selected only if both of them have the same fitness values.

These measures of modularity can also help guide the development of some of the techniques - for example, tag based modules [13, 15] in PushGP - that try to equip the evolutionary processes to evolve more modular programs.

8 CONCLUSIONS

Modularity is all pervasive - from neural networks in our brains to almost all of the human written software - and yet the field of genetic programming has not paid enough attention to it. In this paper, we presented two measures that capture different aspects of modularity. These measures are general purpose and platform independent. We described the procedure to calculate these measures from execution traces. To calculate these measures on some real world programs, we used the programs evolved in PushGP. We have provided a preliminary sketch of ways in which the modularity measures described in this paper could be used to guide evolution to create more modular programs, which we hypothesize will help us to find solutions to more complex problems. These measures can also help us study the effects of modularity on the evolution of software.

REFERENCES

[1] Jeff Clune, Jean-Baptiste Mouret, and Hod Lipson. 2013. The evolutionary origins of modularity. In *Proc. R. Soc. B*, Vol. 280. The Royal Society, 20122863.

[2] Carlos Espinosa-Soto and Andreas Wagner. 2010. Specialization can drive the evolution of modularity. *PLoS computational biology* 6, 3 (2010), e1000719.

[3] George Gerules and Cezary Janikow. 2016. A survey of modularity in genetic programming. In *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 5034–5043.

[4] Thomas Helmuth and Lee Spector. 2015. General program synthesis benchmark suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 1039–1046.

[5] Gregory S Hornby. 2007. Modularity, reuse, and hierarchy: Measuring complexity by measuring structure and organization. *Complexity* 13, 2 (2007), 50–61.

[6] Nadav Kashtan and Uri Alon. 2005. Spontaneous evolution of modularity and network motifs. *Proceedings of the National Academy of Sciences* 102, 39 (2005), 13773–13778.

[7] John R Koza and John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press.

[8] Krzysztof Krawiec and Bartosz Wieloch. 2009. Functional modularity for genetic programming. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM, 995–1002.

[9] Mark EJ Newman. 2004. Fast algorithm for detecting community structure in networks. *Physical review E* 69, 6 (2004), 066133.

[10] Mark EJ Newman. 2006. Modularity and community structure in networks. *Proceedings of the national academy of sciences* 103, 23 (2006), 8577–8582.

[11] Zhenyue Qin, Robert I. McKay, and Tom Gedeon. 2018. Why don't the modules dominate - Investigating the Structure of a Well-Known Modularity-Inducing Problem Domain. *CoRR* abs/1807.05976 (2018). arXiv:1807.05976 http://arxiv.org/abs/1807.05976

[12] Lee Spector. 2001. Autoconstructive evolution: Push, pushGP, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, Vol. 137.

[13] Lee Spector, Kyle Harrington, Brian Martin, and Thomas Helmuth. 2011. WhatâĂŹs in an evolved name? the evolution of modularity via tag-based reference. In *Genetic Programming Theory and Practice IX*. Springer, 1–16.

[14] Lee Spector, Jon Klein, Maarten Keijzer, and Maarten Keijzer. 2005. The push3 execution stack and the evolution of control. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. ACM, 1689–1696.

[15] Lee Spector, Brian Martin, Kyle Harrington, and Thomas Helmuth. 2011. Tag-based modules in genetic programming. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, 1419–1426.