GLEAM: Genetic Learning by Extraction and Absorption of Modules

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

Lee Spector
Amherst College, Hampshire College, UMass Amherst
Amherst, MA
lspector@amherst.edu

ABSTRACT

Efforts to evolve modularity in genetic programming are as old as the field itself. While many techniques have been proposed to evolve programs that make use of simultaneously-evolved modules, the general problem of evolving large-scale, modular software still stands as a central challenge to the field. In this paper, we present GLEAM, a mechanism of achieving modularity, whereby an evolving program uses a local library of modules. The program can refer to the modules during execution and the ones not used by the program are eventually replaced by code segments *extracted* from the program itself. Modules can also be *absorbed* by the program when their references in the program get expanded to the full code segments. We show that this simple mechanism improves the success rate on a number of program synthesis problems.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**;

KEYWORDS

modularity, PushGP, program synthesis, tag-based reference

ACM Reference Format:
Anil Kumar Saini and Lee Spector. 2021. GLEAM: Genetic Learning by Extraction and Absorption of Modules. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3449726.3459544

1 INTRODUCTION

Genetic programming has been successful in synthesizing complex programs, like the ones from the introductory programming textbooks [3]. In order to solve more complex problems, the need for evolving modular programs has long been felt in the genetic programming community. In this paper, we present GLEAM, a simple mechanism whereby evolving programs can create and use modules during evolution. Specifically, each individual has a local library of modules that can be referenced by the program. Handling of the module arguments and return values depends on the underlying system for which GLEAM is being used. The library is updated after every generation (see Section 3.2). Through the experiments presented in this paper, we demonstrate the ability of GLEAM to improve the success rate on multiple benchmark problems.

GECCO '21 Companion, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8351-6/21/07.
https://doi.org/10.1145/3449726.3459544

2 EVOLVING MODULES IN GP

Various attempts have been made to encourage modularity in evolving programs. One of the first attempts include that of John Koza in evolving ADFs [6]. Module Acquisition [1], and others improve upon the concept of ADFs. In Grammatical Evolution (GE), modules are *identified* by looking at the subtrees in the individuals in the population [9] and the underlying grammar is modified to accommodate these modules. A Run Transferable Library [4] is a set of modules that is transferred from one run to the next for a given problem. In tag-based modules [8], a program can label a chunk of code with an integer tag and later refer to it. SignalGP [7] considers programs as collections of functions accessible through identifiers. In Tangled program graphs (TPG) [5], there are two populations, one for *teams* (which act as individuals) and the other for *programs* (which act as modules).

3 GLEAM

The description of GLEAM in this section is for the linear representation of genomes. For other representations, suitable modifications can be made.

To refer to a particular module with identifier i, the program uses the instruction `tagged_i`. To generate tag references, we use a function `tagged_erc_limit`, which, when called, inserts `tagged_j` in the program, where j is an integer chosen randomly between 0 and a pre-defined *limit*. For this paper, this limit is set to 10.

3.1 Initializing the library

In the version of GLEAM that we present here, each individual library contains 10 modules. This is done keeping in mind the memory and computational costs. The modules are initialized using the same instruction set as is available to the program. Consequently, modules, like the program, can also call each other. The underlying system, therefore, should handle any possible recursions due to this. The size of the initial modules was kept one-tenth the size of the initial programs.

3.2 Updating the library

For every generation, in addition to applying various genetic operators on the child genome, its local library is also updated.

3.2.1 Mutation. Every module genome in the library is mutated using the same mutation method used for the child genome. The rate, however, can be different from that of the child genome. This difference in rates will determine whether or not the modules are

Table 1: Genetic Programming Parameters.

Parameter	Value
Population size	1000
Number of generations	300
Parent selection algorithm	lexicase
Mutation operator	UMAD
Mutation rate	0.09
Genome Representation	Plushy
Number of runs per condition	50

being *protected*, i.e., the program is keeping important segments from changing too frequently.

3.2.2 Extraction. A module is considered as being *used* if it is called by the program directly or indirectly (it is called by a module which is being called by the program). In the extraction operator, every unused module gets replaced by a new module with a certain probability. Various ways to search for the new modules are: randomly generated code segment, randomly chosen segment of the host genome or any other genome in the population, another module from the same individual or from another individual in the population, etc. To sample the lengths of the new modules, we use the Negative Binomial distribution with parameters 2 and 0.4.

3.2.3 Absorption. In this operation, every module reference, with a certain probability, be it in the program or the modules themselves, gets replaced by the code segment it refers to.

4 EXPERIMENTS

Experiments were conducted in a GP system called PushGP (the version called Clojush[1]). To test the effectiveness of GLEAM, we ran it on six problems: Last Index of Zero, Count Odds, Sum of Squares, Small or Large, Compare String Lengths, Greatest Common Divisor. The descriptions of first five problems are given in the General Programming Benchmark Suite [3]. The last one is a commonly used problem: Given two integers, the solution should return their greatest common divisor. Various GP parameters are given in Table 1. GLEAM specific parameters and their values are given below. There is no mutation at the level of modules. Every unused module is replaced with a probability of 0.75 by a randomly chosen sequence of genes that is extracted from the host genome. The lengths of new modules are drawn from a Negative Binomial distribution. We allow module references to get expanded in the program as well as in modules with a probability of 0.1.

4.1 Results

Table 2 gives the number of successes for various problems. To qualify as a solution, the program must have a zero error on the training set, which was used during evolution, and another held-out test set, which was not used during evolution. The programs which produced zero error on the training set were first simplified (procedure detailed in [2]), before running them on the test set.

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

Table 2: Number of success out of 50 for various configurations.

Problem	W/out GLEAM	With GLEAM
Last Index of Zero	29	37
Count Odds	3	5
Sum of Squares	9	6
Greatest Common Divisor	17	20
Small or Large	4	6
Compare String Lengths	14	12

Clearly, GLEAM appears to improve the success rate for some of the problems from the benchmark suite.

5 CONCLUSIONS

We introduced a general framework for evolving modules in GP called GLEAM. In this paper, we implement it in PushGP and show that it improves the success rate on multiple problems from the program synthesis benchmark suite.

For problems more complex than the ones studied here, the utility of GLEAM remains to be seen. Additionally, why GLEAM improves the success rate for some problems while not for others can be taken up in a future study.

ACKNOWLEDGMENTS

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.

This work was performed in part using high performance computing equipment obtained under a grant from the Collaborative R&D Fund managed by the Massachusetts Technology Collaborative.

REFERENCES

[1] Peter J Angeline and Jordan B Pollack. 1992. The evolutionary induction of subroutines. In *Proceedings of the fourteenth annual conference of the cognitive science society*. Bloomington, Indiana, 236–241.
[2] Thomas Helmuth, Nicholas Freitag McPhee, Edward Pantridge, and Lee Spector. 2017. Improving generalization of evolved programs through automatic simplification. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 937–944.
[3] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. July (2015), 1039–1046. https://doi.org/10.1145/2739480.2754769
[4] Maarten Keijzer, Conor Ryan, and Mike Cattolico. 2004. Run transferable libraries—learning functional bias in problem domains. In *Genetic and Evolutionary Computation Conference*. Springer, 531–542.
[5] Stephen Kelly, Jacob Newsted, Wolfgang Banzhaf, and Cedric Gondro. 2020. A modular memory framework for time series prediction. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. 949–957.
[6] John R Koza and John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press.
[7] Alexander Lalejini and Charles Ofria. 2018. Evolving event-driven programs with SignalGP. *arXiv* (2018), 1135–1142.
[8] 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*. 1419–1426.
[9] John Swafford, Miguel Nicolau, Erik Hemberg, Michael O'Neill, and Anthony Brabazon. 2012. Comparing methods for module identification in grammatical evolution. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. 823–830.