Towards Knowledge-guided Genetic Improvement

Oliver Krauss*†, Hanspeter Mössenböck*, Michael Affenzeller†
{oliver.krauss,michael.affenzeller}@fh-hagenberg.at,hanspeter.moessenboeck@jku.at
*Johannes Kepler University
Altenbergerstrae 69
Linz, Austria 4040
†University of Applied Sciences Upper Austria
Softwarepark 13
Hagenberg, Austria 4232

ABSTRACT

We propose Knowledge-guided Genetic Improvement as a combination of Grammar-guided Genetic Programming with Tree-based Genetic Programming. Instead of utilizing a grammar directly, an operator graph based on that grammar is created, that is responsible for producing abstract syntax trees. Each operator contains knowledge about the grammar symbol it represents and returns only trees valid according to user-defined restrictions such as depth, complexity and approximated run-time performance.

The expected benefits are a search space that excludes invalid individuals in an evolutionary run, ensuing a reduced overhead to evaluate invalid solutions and improving overall quality of the explored search space. The operator graph supports improvements based on previously run experiments and extensions towards further non-functional features.

CCS CONCEPTS

•**Software and its engineering** →**Source code generation;** Interpreters; •**Computing methodologies** →*Genetic algorithms;*

KEYWORDS

search space, genetic improvements, evolutionary operators, performance approximation

ACM Reference format:
Oliver Krauss*†, Hanspeter Mössenböck*, Michael Affenzeller†. 2020. Towards Knowledge-guided Genetic Improvement. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 2 pages.
DOI: 10.1145/1122445.1122456

1 INTRODUCTION

We propose a combination of Tree-based Genetic Programming with Grammar-guided Genetic Programming to be used in Genetic Improvement (GI) by restricting the search space with knowledge about non-functional features of the programming or language being used. Knowledge is defined as measurement of non-functional features such as branch complexity, run-time performance or code size, as well as the relations (if - condition - then - else) and required order between concepts of a language (memory allocation - write - read sequences).

Grammar-guided Genetic Programming (GGGP) is primarily directed towards improving the original crossover operator designed by Koza [4] to only consider crossover points that are syntactically correct according to a provided grammar [5]. Tree Genetic Programming (TGP) utilizes a tree structure, often an Abstract Syntax Tree (AST), as a representation for its individuals. The AST representation enables the utilization of useful operators. One example is the homologous crossover [1]. GGGP and TGP have also been previously used in combination resulting in Tree-adjunct Grammar Guided Genetic Programming (T3GP) [2, 3], combining the advantages of both approaches, utilizing the tree representation for the operators enriched with the syntactic information available from the grammar.

Our work proposes the combination of GGGP and TGP in a different way, turning the grammar itself into an operator graph that is both responsible for selection and creation of AST individuals, which remains the representation choice for individuals in the population. These individuals will always be generated in the valid non-functional search space of the GI experiment, such as maximal run-time performance, depth or code size. The operator graph can be utilized in all three major genetic operators — create, crossover, mutate — and can be modified during and after running GI experiments to further restrict the search space and improve the success rate of created trees. Our approach, unlike GGGP which uses context free grammars, can also utilize the context of individuals to improve individual creation even further.

2 METHODS

We propose an operator graph where every single construct in a grammar has a corresponding operator (see Figure 1). Starting from an root operator, which links to all operators, every operator maps to either the root operator in case no restrictions are defined, or alternatively maps to a specialized operator subgraph restricting its use. In the create and mutate operations the root operator is responsible for selecting only such operators that will create valid nodes according to user-defined restrictions such as a subtree valid at a specific position in a function. Crossover uses the operator

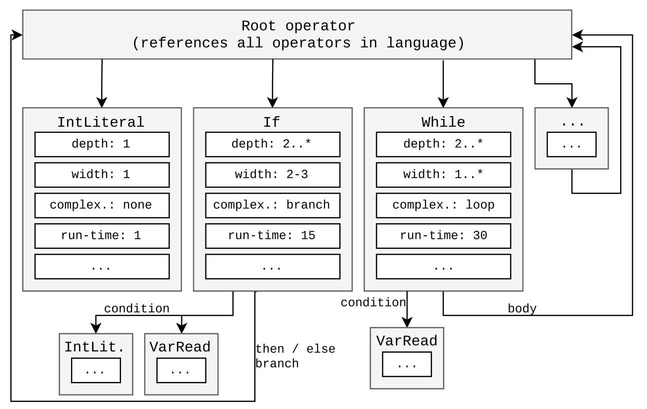

Figure 1: Operator graph. Nodes (grey) represent concepts in a language and contain knowledge about these concepts (white). They contain edges to child nodes.

graph for selection of all valid subtrees in tree A that can replace a selected crossover point in tree B.

Knowledge is encoded in each operator and the edges between the operators. Whenever a new individual is created using an operator, the operator will add its minimal values to a request (ex. minimal depth) and ask the operators at each child edge (see Figure 1) if they can create a valid subtree. Only valid child edges (maximum depth not exceeded) can then be selected to create an individual. A selected child operator will repeat that process with its own children. These individuals will always be generated in the valid non-functional search space of the GI experiment, such as complexity according to McCabe[6], maximal assumed run-time performance, depth or code size. To measure the assumed run-time performance each operator is assigned an assumed weight. The assignment happens by benchmarking every concept in the grammar. For if branches, an assertion is taken which branch is more likely to be selected. For loops, an assertion is taken how often the loop is repeated. While this does not return an accurate measure, due to compiler optimizations reducing the runtime below the assumed performance, it does represent a valid maximum. These assumptions can be updated to real measurements when a generated individual is tested.

An unmanaged operator graph will reference from the root operator to each concept in the grammar. Each concept in turn will reference the root operator (Figure 1: then branch, else branch, body). To reduce incorrect individuals the edges in the operator graph can be pruned or redirected. For example, in Figure 1 the while loop condition edge excludes any literal node (int, double, char) from being used as this would result in endless loops or dead code that is never called.

Each node can also contain additional business logic. For example, read operation nodes check if there is a corresponding write operation to the same Stack/Heap field by another node in the AST. This prevents individuals failing their execution due to data access violations.

3 PROPOSED BENEFITS

Our approach increases the percentage of valid generated trees in an evolutionary search. The primary reason for this is the knowledge contained in the operator graph concerning the language semantics, such as needed allocate-write-read sequences. This allows the restriction of read operations only on validly initialized variables. The application of knowledge about non-functional features such as the assumed run-time performance, prevents the generation of individuals that will not represent an improvement in the non-functional domain.

Another promising area is the reduction of invalid AST solutions occurring during compile time, such as incorrect function calls, array index violations and endless-loops. This is achieved by pruning or redirecting edges between nodes in the operator graph, and can happen in preparation of, or during a GI experiment using the operator graph. For example, in the domain of endless loops, excluding the option of literal values in the condition of loops reduces the amount of individuals that have to be aborted due to endless execution.

The drawback of knowledge-guided genetic improvement is the amount of work required to provide that knowledge, such as creating benchmarking operations for the assumed run-time performance or static analysis to determine branching complexity. If the knowledge in the operator graph is incorrect it presents a threat to validity for conducted GI experiments.

4 CONCLUSION AND FUTURE WORK

Utilizing an operator graph to create individuals in GI runs is promising, because it allows restricting the search space to manageable levels. It improves the amount of valid abstract syntax trees generated, and shortens the time required to find improvements, as more valid solutions will be explored.

The next step with this approach is to test operator graphs on problems described in literature to compare the quality of the population and its diversity to other approaches.

REFERENCES

[1] F. D. Francone, M. Conrads, W. Banzhaf, and P. Nordin. 1999. Homologous Crossover in Genetic Programming. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2* (Orlando, Florida) *(GECCO'99)*. Morgan Kaufmann Publishers Inc., 1021–1026.

[2] N.X. Hoai and R.I. McKay. 2001. *A Framework for Tree-adjunct Grammar Guided Genetic Programming.* Technical Report. Canberra, Australia. 93–100 pages.

[3] N.X. Hoai, R.I. McKay, D. Essam, and R. Chau. 2002. Solving the Symbolic Regression Problem with Tree-Adjunct Grammar Guided Genetic Programming: The Comparative Results. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, Vol. 2. 1326–1331 vol.2. https://doi.org/10.1109/CEC.2002.1004435

[4] J. R. Koza. 1990. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems.* Technical Report. Stanford, CA, USA.

[5] D. Manrique, J. Rios, and A. Rodríguez-Patón. 2009. Grammar-Guided Genetic Programming. In *Encyclopedia of Artificial Intelligence.* https://doi.org/10.4018/978-1-59904-849-9.ch114

[6] T. J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Software Eng.* SE-2, 4 (Dec 1976), 308–320. https://doi.org/10.1109/TSE.1976.233837