Mining Patterns from Genetic Improvement Experiments

Oliver Krauss*†, Hanspeter Mössenböck*, Michael Affenzeller†
*Johannes Kepler University
Linz, Austria
Email: hanspeter.moessenboeck@jku.at
†University of Applied Sciences Upper Austria
Wels, Austria
Email: oliver.krauss, michael.affenzeller@fh-hagenberg.at

***Abstract*—When conducting genetic improvement experiments, a large amount of individuals (≈ population size * generations) is created and evaluated. The corresponding experiments contain valuable data concerning the fitness of individuals for the defined criteria, such as run-time performance, memory use or robustness. This publication presents an approach to utilize this information in order to identify recurring context independent patterns in abstract syntax trees (ASTs). These patterns can be applied for restricting the search space (in the form of anti-patterns) or for grafting operators in the population. Future work includes an evaluation of this approach, as well as extending it with wildcards and class hierarchies for larger and more generalized patterns.**

***Keywords*-Genetic Improvement; Abstract Syntax Tree; Pattern Mining; Frequent Subgraph Mining**

I. Introduction

A recurring challenge in Genetic Improvement (GI) of programs, is that a large part of the population fail to compile, or alternatively produces undesirable behavior, such as endless loops [1], [2], [3], [4]. Identifying anti-patterns could help to restrict the search space during the creation or modification of individuals, to increase the amount of valid solutions in a population, as well as reducing the run time of the GI run itself, since solutions that don't compile won't be explored.

Grafting operators usually transplant code from other parts of the same program or from input defined by a human [5], [6]. Recurring parts of solutions with a high quality could be identified as grafting patterns, and extend the current grafting operator approach.

II. Approach

Our approach for pattern mining extends an existing GI framework that uses Abstract Syntax Trees (AST) as a representation [7]. Any existing framework could be extended by this approach by logging the following data into a graph database:

- The original program(-part) to be optimized
- Experiment metadata, such as mutation probability etc.
- Language metadata, like existing operators and operands
- Attempted optimizations (single or multi-objective)
- Every individual in the population, including its fitness evaluations
- The relationships between individuals (e.g. graft, crossover, mutation, ...)

From recordings of multiple GI experiments, Frequent Subgraph Mining (FSM) is then applied to identify potential candidates for patterns. This works by first selecting the appropriate domain of graphs that should be mined.

To mine *anti-patterns*, trees are selected that have a low quality. In the case of test-driven GI, ASTs are selected that fail all tests in the suite, or don't compile at all.

Optimization-patterns are intended for use in grafting. There are multiple options to mine these patterns. The simplest option is to mine patterns in one specific program, but to get more generalizable patterns other options are to mine all graphs which perform well over all parameters in the pareto front, or graphs that perform similarly on one parameter. Additionally, the domain can be extended beyond a single program by selecting the domain through the similarity of input and/or output data in the applied test suites.

After selecting the search space the identified trees are normalized. This includes numbering of variables to avoid missing patterns due to different naming schemes of variables. The unnecessary bloat, that often occurs in GI and genetic programming [8], contains frequently recurring patterns that are misidentified as highly relevant. Trees are pruned beforehand, to prevent identifying patterns from bloat. In the future, we also plan to mine pruning-patterns which needs to omit this normalization step to find prunable paths.

After preparing the appropriate search space, an adaption of the SLEUTH algorithm for mining frequent sub trees is used to identify potential candidates for patterns [9]. Candidates are all frequently recurring sub trees in the search population. To avoid the overestimation of smaller trees, which often occur multiple times in the same AST (e.g. access to a variable usually recurs multiple times in the same function), two separate metrics exist. One metric only counts in how many ASTs the subgraph occurs, whereas the other counts all occurrences in all ASTs. According to the *apriori rule*, which states that for any frequent graph all its subgraphs are also frequent, sub ASTs are pruned and only the largest frequent ASTs are considered as patterns. This is similar to the *term frequency to inverse document frequency* (tf-idf), which is used in text mining to determine the importance of terms [10].

From this point, frequently occurring ASTs are manually analyzed for the applicability as an anti-pattern or optimization-pattern. This is currently being done in a subset of the C language, as well as the JavaScript language. As of the time of writing by using this approach of pattern mining several anti-patterns were identified. These include accessing function parameters out of bounds and incorrectly accessing global objects.

These patterns are currently applied manually. The optimization-patterns are used as grafts during the creation of the initial population, as well as in a grafting mutator. The anti-patterns are grouped into a hierarchy that is used for the creation of (sub) ASTs in the mutator or at the initial creation of the population. This hierarchy prunes the options available during the creation process, as the AST creation descends the tree.

III. Challenges and Future Work

The next step to improve our approach, is the introduction of pattern wildcards to combine smaller patterns into larger ones. These wildcards attempt to link multiple patterns structurally over the (*) wildcard, meaning any number of nodes in-between, the (.) wildcard meaning any single node, the (?) wildcard, meaning one or no node. These wildcards are currently still being experimented with, as they increase the search space for patterns, and thus often make the search unfeasible.

Patterns can be generalized over the type hierarchy of the nodes occurring in the ASTs. For example, an *integer literal node* can be generalized to an *integer node* or even further to a *numerical node*. This is attempted through a class hierarchy that is defined for every operator and operand that the GI experiments can use.

Both, the wildcards and the extension of patterns towards a class hierarchy increase the complexity how the patterns can be applied, as they can't be simply grafted into an existing tree anymore. For the creation of a tree, the patterns can simply create something matching the wildcards. However, for mutation the question is, if patterns should just be used to create new sub ASTs, or if they should be partially matched onto the wildcards. The effects of these approaches will be analyzed in the future.

Another currently open challenge is that mined patterns are still evaluated for feasibility manually. A possible approach to partially automate pattern identification, is automatically applying pattern candidates in experiments, and to test if they behave as expected. For anti-patterns this could be as simple as randomly generating trees around the anti-patterns and verifying that the tree fails. For optimization-patterns this verification is much more complex. One possible solution is to re-run experiments that contained ASTs in which a pattern candidate was identified and to apply the pattern in a grafting operator in order to check if this has a positive effect on finding better solutions. This check could be done by comparing the amount of evaluations needed until a good solution is found or by checking if the average quality of the populations increases.

In the future, we will also attempt to define transformation patterns. This means that the patterns themselves should be applicable to a given AST instead of being used simply as grafts or anti-patterns. Transformation patterns require the identification of a pattern in the source and target domain and matching those two patterns to create a transformation pattern. This means that if an AST matches a source pattern, it can be transformed into the target pattern. The transformation requires the use of wildcards, to find out which nodes are transferred from the applicable AST into a new AST, and which ones are replaced or modified. For GI, transformation patterns can be useful for pruning, or as a form of mutator operator. In addition, this could lead to generally applicable optimization-patterns, that can be used on their own.

IV. Discussion and Outlook

This paper outlines an early-stage approach to mine and utilize the data produced during GI experiments. The mining of patterns as well as their application to prune the search-space and to use them for grafting is currently being done, but a study on the effectiveness of the approach is still needed. This approach will likely benefit from the use of wildcards and additional class hierarchies for finding and creating patterns, altough its feasibility still has to be evaluated.

Our approach is currently only applicable for GI frameworks that use an AST-based representation for optimization. Many frequent subgraph mining algorithms exist that allow mining on graphs, and switching to one, would easily adapt our approach towards graph-based representations. On a conceptual level, pattern mining could be used in any representation, but requires a completely different approach for mining if they are not graph based.

References

[1] M. Orlov and M. Sipper, "Genetic programming in the wild: Evolving unrestricted bytecode," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 1043–1050. [Online]. Available: http://doi.acm.org/10.1145/1569901.1570042

[2] ——, "Flight of the finch through the java wilderness," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 2, pp. 166–182, April 2011.

[3] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, *Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 137–149.

[4] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 118–135, Feb 2015.

[5] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 306–317. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635898

[6] W. B. Langdon and R. Lorenz, "Improving sse parallel code with grow and graft genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '17. New York, NY, USA: ACM, 2017, pp. 1537–1538. [Online]. Available: http://doi.acm.org/10.1145/3067695.3082524

[7] O. Krauss, "Towards a Framework for Stochastic Performance Optimizations in Compilers and Interpreters: An Architecture Overview," in *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ser. ManLang '18. New York, NY, USA: ACM, 2018, pp. 9:1–9:7. [Online]. Available: http://doi.acm.org/10.1145/3237009.3237024

[8] S. Luke and L. Panait, "A comparison of bloat control methods for genetic programming," *Evol. Comput.*, vol. 14, no. 3, pp. 309–344, Sep. 2006.

[9] M. J. Zaki, "Efficiently mining frequent embedded unordered trees," *Fundamenta Informaticae*, vol. 66, no. 1-2, pp. 33–52, Mar/Apr 2005, special issue on Advances in Mining Graphs, Trees and Sequences.

[10] F. D. Malliaros and K. Skianis, "Graph-based term weighting for text categorization," in *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*, ser. ASONAM '15. New York, NY, USA: ACM, 2015, pp. 1473–1479.