Hybridizing Evolutionary Algorithms and Clustering Algorithms to Find Source-Code Clones

Andrew Sutton, Huzefa Kagdi, Jonathan I. Maletic, L. Gwenn Volkert
Department of Computer Science
Kent State University
Kent Ohio 44242

{asutton, hkagdi, jmaletic, volkert}@cs.kent.edu

ABSTRACT

This paper presents a hybrid approach to detect source-code clones that combines evolutionary algorithms and clustering. A case-study is conducted on a small C++ code base. The preliminary investigation indicates that such an approach is effective in detecting groups of source-code clones.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Restructuring, reverse engineering, and reengineering, I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search.

General Terms

Algorithms, Design, Experimentation.

Keywords

Evolutionary Algorithms, Software Engineering, Clone Detection.

1 INTRODUCTION

In software a clone is defined as a unit of source code that is identical or similar to another unit of source code. Software clones are classified as exact and near-miss clones [1]. Exact clones are pure replicas of exact textual lines, whereas near-miss clones are those with a certain variation in the textual, syntactic, and semantic composition. A number of approaches exist in software engineering literature addressing identification of both exact and near-miss clones. These techniques range from simple lexically-based [1, 4, 6] to complex structural and/or semantics similarity analysis [2, 7, 8].

Evolutionary Algorithms (*EA*s) are typically applied to problems for which the search space is too large to investigate exhaustively. One can view software engineering as a search problem [11] however, there are issues and challenges in adapting evolutionary computing to this domain [3, 5]. Nonetheless, evolutionary computing has been applied to various software engineering activities including software testing [9] and reverse engineering [10].

In the work presented here, the contribution is twofold. On the evolutionary computing front, this is an effort to extend its scope and application within the domain of software engineering, which to date has focused more on testing issues. To this end, we propose an *EA*-based search technique for identifying clones in source code. In the context of software engineering, this is a non-traditional method for identifying near-miss textual clones. On the software engineering front, the eventual goal of the presented technique is to obtain a solution that gives the smallest number of clone-groups such that the similarity between clones and the length of an individual clone within each group is maximized.

GECCO'05, June 25-29, 2005, Washington, DC, USA.
ACM 1-59593-010-8/05/0006.

2 THE APPROACH: EA AND CLONES

In any given software system the number of clone-groups and number of clones in each group is not known *a priori* and is dependent on the type of clone being searched for (e.g. exact or near-miss clones). Given that our goal is to find a set of clone groups we develop a method for an EA inspired search with a dynamic population size. Each individual represents a candidate clone group and thus the goodness of the solution is determined at the level of the population (i.e. minimizing the number of individuals while maximizing the similarity of the clones represented within each individuals). This type of *EA* is more easily implemented with the evolutionary programming (*EP*) paradigm since both the population and the individuals will be dynamically changing in size through-out the evolutionary process. Detailed specification of our approach is defined through the following components.

Individual Representation: Individuals are represented by a variable–sized vector. The individual $\bar{x}$ consists of the genetic component $\bar{c}$ and the strategy parameters s and l. The genetic component of an individual represents a clone-group. Each gene (i.e., code fragment) of an individual is represented by a starting location s_i and the length l_i.

$$\bar{x} = \langle \bar{c}, s, l \rangle, \quad \bar{c} = \langle c_1, c_2, \ldots c_n \rangle, c_i = \langle s_i, l_i \rangle$$

Initialization by Clustering: We use an agglomerate clustering algorithm to form (initialize) a population of individuals on the basis of their genetic composition. Notice that in the representation of the code fragments, the phenotype (i.e., the actual source code content) is not directly encoded in the genotype; rather the individual genes behave as "pointers" to the phenotype. We use a similarity metric based on longest common subsequence (*LCS*) for comparing the textual contents pointed to by the genes as indicated by their allele values.

Variation Operators: Like traditional *EP*s, this algorithm implements no recombination operator, relying exclusively on mutation operators for introducing modified individuals into the population. We define a set of mutation operators which allows us to adjust the starting point (*Shift*), length of each clone in an individual (*Grow*), and both (*Kick*). We use a die rolling approach to determine which mutation operator is executed.

Parent and Survivor Selection: These mechanisms are slightly different than traditional *EP*s due to a variable population size and a lack of absolute fitness values. Every individual in the current population is selected as a parent. Initially, each parent creates an identical child that is then subjected to mutation and re-clustering. The re-clustering may result in individuals that are no better than the parents (e.g., unsuccessful *Grow* mutation) or no relation between parents and children (e.g., successful *Kick* mutation). The entire current population is not allowed to pass to the next generation. A culling step is employed to eliminate individuals that are entirely contained in other individuals. Thus, the population growth is controlled after each generation.

Termination: We provide two forms of termination for this algorithm: generation-based and stability-based methods. The stability-based approach terminates the algorithm once a specified number of generations have elapsed without any improvement in the population.

Initialization: The initial population is formed by placing each code fragment in its own cluster. A hashing algorithm is used to create an initial set of clusters of the given code fragments.

Pre- and Post- Processing: We implemented a pre-processing filter to remove "noise" such as white space and trivial code fragments with a little or no significance as candidate clones. Furthermore, we added a post-processing stage to optimize the results (e.g., merging clusters consisting of adjacent lines of source code).

3 CASE-STUDY

We conducted a preliminary investigation to determine the applicability and effectiveness of our hybrid algorithm in detecting clones at the file and system levels. We selected the (C++) source code of our algorithm as a test system. It consists of 25 files and approximately 2600 textual lines of code (LOC). We studied three performance metrics, namely, the number of clones (groups), the maximum clone size, and the average clone size. We carried out a number of runs of the algorithm and recorded results for various combinations of parameters. We validated the results manually via spot checks and learned that the algorithm worked with a reasonable precision.

4 CONCLUSIONS

We have mapped the clone detection problem into a search representation that can be addressed with a hybrid approach that combines an evolutionary algorithm and clustering algorithm. We believe that the hybrid evolutionary-clustering approach provides a promising solution to problems of this type. The preliminary investigations imply that this approach successfully finds (with some degree of error) clone classes by continually adjusting and re-clustering candidates to maximize their similarity content while simultaneously minimizing the number of clone groups. While the algorithm may not be as time-efficient as others, we conjecture that it will outperform other traditional approaches in terms of recall and precision. In the future, we will continue to revise the algorithm, experimenting with different representations, clustering algorithms, similarity measures, and additional heuristics. Moreover, we are in the process of conducting a case study investigating this approach on a much larger body of software (e.g., the Linux kernel or a KDE release).

5 REFERENCES

[1] Baker, B., "On Finding Duplication and Near-Duplication in Large Software Systems", in Proceedings of Working Conference on Reverse Engineering, Toronto, Ontario, Canada, July 1995, pp. 86-95.

[2] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L., "Clone detection using abstract syntax trees", in Proceedings of International Conference on Software Maintenance, Bethesda, Maryland, November 16-19 1998, pp. 368-377.

[3] Clark, J., Dolado, J. J., Harman, M., Hierons, R., Jones, B. F., Lumkin, M., Mitchell, B. S., Mancordis, S., Rees, K., Roper, M., and Shepperd, M., "Reformulating Software Engineering as a Search Problem", Journal of IEE Proceedings - Software, vol. 150, no. 3, 2003, pp. 161-175.

[4] Ducasse, S., Rieger, M., and Demeyer, S., "A Language Independent Approach for Detecting Duplicated Code", in Proceedings of International Conference on Software Maintenance, Oxford, England, August 30 - September 3 1999, pp. 109-118.

[5] Harman, M. and Jones, B. F., "Search-Based Software Engineering", Information and Science Technology, vol. 43, no. 14, December 2001, pp. 833-839.

[6] Kamiya, T., Kusumoto, S., and Inoue, K., "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code", Transactions on Software Engineering, vol. 28, no. 7, July, 2002, pp. 654-670.

[7] Komondoor, R. and Horwitz, S., "Finding duplicated code using program dependences", in Proceedings of European Symposium on Programming, Genoa, Italy, April 2-6 2001.

[8] Marcus, A. and Maletic, J. I., "Identification of High-Level Concept Clones in Source Code", in Proceedings of Automated Software Engineering (ASE'01), San Diego, CA, November 26-29 2001, pp. 107-114.

[9] Michael, C., McGraw, G., and Schatz, M., "Generating Software Test Data by Evolution", IEEE Transactions on Software Engineering, vol. 12, Dec. 2001, pp. 1085-1110.

[10] Mitchell, B. S., Mancordis, S., and Traverso, M., "Search-Based Reverse Engineering", in Proceedings of International Software Engineering and Knowledge Engineering Conferences (SEKE'02), Ischia, Italy, July 2002, pp. 431-438.

[11] Simon, H., "Whether Software Engineering Needs to be Artificially Intelligent"", IEEE Transactions on Software Engineering, vol. 12, no. 9, September 1986, pp. 726-732.