Towards a multi-scale approach for source code approximate match report

Michel Chilowicz Etienne Duris Gilles Roussel
Université Paris-Est
Laboratoire d'Informatique de l'Institut Gaspard-Monge, UMR CNRS 8049
5 Bd Descartes — 77454 Marne-la-Vallée Cedex 2 — France
firstname.lastname@univ-paris-est.fr

ABSTRACT

Finding exact clones in source code can be efficiently handled using classical exact substring or subtree pattern matching techniques inspired from genomics applications. These methods may be wisely employed as a foundation to sketch new techniques highlighting duplicated code chunks presenting minor edits or more extensive modifications at a higher structural scale. The main goal is to improve recall of small near matches and to aggregate them into larger ones to provide a more global view of similarities with a reasonable complexity. These concerns are essential to be able to address a large database of source code projects.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; I.5.3 [**Computing Methodologies**]: Pattern Recognition

General Terms

Algorithms

Keywords

Source code similarity, clones, software plagiarism

1. CONTEXT

Most popular clone detection tools, either in a code reengineering or in a plagiarism highlighting context, consider sequences of tokens [10, 8] or syntax trees [1] as an abstracted view of the source code. For token sequence representations, groups of occurrences of exact repeated token factors can be found with suffix indexing structures like the suffix arrays [9] or the suffix trees [13] in linear time. Zones of high density of similar n-grams [12] can also be used to spot similarities. Nevertheless, when local edits are introduced between code clones, looking for exact repeated factors is inappropriate: in this case, local alignment algorithms [6] could be used to retrieve approximate substring matches containing small unmatched gaps. However, these techniques require a detrimental quadratic running cost in number of tokens: it is preferable to use them on carefully selected zones containing suspected clones.

IWSC2010 May 8, 2010, Cape Town, South Africa

2. FUNCTIONAL CALL GRAPHS OF TOKEN SEQUENCES

To cope with multiple occurrences of similarities and to have a global view of them, we proposed in [3] a method allowing similarity report at a kind of *function* level. Using suffix structures, we split each original function of the source code into different components that are either token substrings shared by other functions or unmatched token sequences. Shared token substrings may themselves be decomposed with the help of smaller nested similarities found among them. Following this approach, we transform individual function call graphs of the compared projects into a global call graph introducing new synthetic functions that represent the granularity of shared chunks of code across the projects. We do not use these call graphs for refactoring purpose, but rather to define several metrics based on the amount of code represented by shared nodes between projects. This method allows similarity retrieval at a synthetic-function level despite extensive edits involving removal, transposition, addition of source code or even inlining or outlining of functions (clones of type 3 according to [2] or types 3-4 for [11]).

Thanks to this factorization approach, comparing elementary unmatched token sequences using an alignment algorithm allows retrieval of similarities that would be missed otherwise thanks to resiliency towards local edits (like removing or adding neutral token sequences, for instance - 0 or * 1 in arithmetic expressions like in figure 1). Identifying tiny local similarities may also help to counter to the development of identifiers into expressions (like the development of identifier diff seen in lines 8-9 of stdDev2) but also to select pairs of token sequences containing high densities of tiny clones that will be tested through alignment methods.

3. CLASSIFYING ABSTRACT SYNTAX SUBTREES

Concerning methods based on syntax tree representations, most accurate approaches [14] involve extensive and very costly dynamic programming comparison of all subtrees comparing to linear token approaches. Preclustering potential similar subtrees, using degraded hashing strategies or metrics [7], appears essential to reduce the number of subtree

```
1  double mean(double[] data) {
2    double sum = 0.0;
3    for (double datum: data) sum += datum;
4    return sum / data.length;
5  }
6  double stdDev1(double[] data) {
7    double stddev = 0.0;
8    double mean = mean(data);
9    for (double datum: data) {
10     double diff = datum − mean;
11     stddev += diff ∗ diff;
12   }
13   return Math.sqrt(stddev);
14 }
```

```
1  double stdDev2(double[] donnees) {
2    double somme = 0.0;
3    for (double donnee: donnees)
4      somme = somme + donnee;
5    double moyenne = somme / donnee.length − 0;
6    double ecartype = 0.0;
7    for (double donnee: donnees)
8      ecartype += (donnee − moyenne)
9        ∗ (donnee − moyenne) ∗ 1;
10   return Math.sqrt(ecartype);
11 }
```

Figure 1: An original standard deviation computation function stdDev1 **in Java and a transformed copy** stdDev2 **involving inlining, identifier renaming and local edits**

comparisons.

We explore in [4] some of these hashing strategies based on an abstracted view of the syntax tree associated with different hash functions. We could then introduce hash values of subtrees according to several abstraction profiles (confounding all types, discarding small subtrees above a given size, considering commutativity of operators...) and infer similarity metrics on subtree pairs according to these abstraction profiles. Thus, duplicated subtrees involving most trivial local edits are detected and their distance quantified.

4. STRUCTURAL CONSOLIDATION OF LOCAL CLONES

The second challenge related to clone detection based on syntax trees focuses on merging simple subtree clones to form higher level clones. This problem does not involve only gathering sibling subtree clones that can be handled with suffix indexation algorithms [5] but also more distant clone subtrees like cousins. It allows detection of large clones composed of smaller ones that are transposed or flooded with other clones that are too heavily transformed to be recognized.

Gathering close clones in the syntax tree using various fast heuristics is an interesting issue to be explored. The call graph can also be used to assemble clones that are potentially reachable from a common function. It leads then to a more concise view of the similarities in source code at a higher level that may be later refined according to the user will. For example, in figure 1, reporting a single match between the complete functions stdDev1 and its copied counterpart stdDev2 appears more appropriate that two matches involving mean and stdDev1 respectively linked to the start and the end of stdDev2.

5. REFERENCES

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98*, page 368, Washington, DC, USA, 1998. IEEE CS.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[3] M. Chilowicz, E. Duris, and G. Roussel. Finding similarities in source code through factorization. In *LDTA'08*, volume 238(5) of *ENTCS*, pages 47–62, Budapest, Hungary, Apr. 2008.

[4] M. Chilowicz, E. Duris, and G. Roussel. Syntax tree fingerprinting: a foundation for source code similarity detection. Technical Report 2009-03, LIGM Université Paris-Est, 2009.

[5] M. Chilowicz, E. Duris, and G. Roussel. Syntax tree fingerprinting for source code similarity detection. In *ICPC'09*, pages 243–247, Vancouver, BC, Canada, May 2009. IEEE CS.

[6] R. Irving. Plagiarism and collusion detection using the Smith-Waterman algorithm, 2004.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07*, pages 96–105, Washington, DC, USA, 2007. IEEE-CS.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[9] U. Manber and G. Myers. *Suffix arrays: a new method for on-line string searches*. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 1990.

[10] L. P. Prechelt, U. Karlsruhe, and G. Malpohl. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8:1016–1038, 2000.

[11] C. K. Roy and J. R. Cordy. Scenario-based comparison of clone detection techniques. In *ICPC*, pages 153–162. IEEE, 2008.

[12] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003*, pages 76–85. ACM Press, 2003.

[13] P. Weiner. Linear pattern matching algorithm. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.

[14] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.