Syntax tree fingerprinting for source code similarity detection

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
Email: firstname.lastname@univ-paris-est.fr

Abstract

Numerous approaches based on metrics, token sequence pattern-matching, abstract syntax tree (AST) or program dependency graph (PDG) analysis have already been proposed to highlight similarities in source code: in this paper we present a simple and scalable architecture based on AST fingerprinting. Thanks to a study of several hashing strategies reducing false-positive collisions, we propose a framework that efficiently indexes AST representations in a database, that quickly detects exact (w.r.t source code abstraction) clone clusters and that easily retrieves their corresponding ASTs. Our aim is to allow further processing of neighboring exact matches in order to identify the larger approximate matches, dealing with the common modification patterns seen in the intra-project copy-pastes and in the plagiarism cases.

1 Introduction

Software analysis, maintenance and reengineering could often benefit from performing clone detection, either for software evaluation or for refactoring issues [1, 5, 16]. Several tools address the problem of identifying software clones that come from copy-paste modifications sometimes followed by slight modifications [4, 20]. A related more complex problem is the detection of software plagiarism, i.e. copies with intentional obfuscations, for instance for license issues or in evaluation of student projects [17, 8]. Even if they could rely on similar transformations, tools addressing this latter issue often require more complex abstraction and processing than for simple clone detection. They also support a loosened association of code chunks [24, 18, 6].

To bring these two problems together into the general issue of source code similarity detection, the choice of an appropriate representation for the source code is required. Four kinds of representation are widely used by similarity detection tools: translation of source in a vector space using metrics (easily defeated by trivial edits), token sequence linear representation, AST representation obtained by parsing or PDG representation. Linear representations by (abstracted) token sequences of source code allow the comparison algorithms to be efficient (based on global [19] or local alignment [10] and n-gram fingerprinting [24, 21]), but they hide the structural organization of the program. Richer representations (ASTs or PDGs) permit fine-grained analysis, manipulation and comparison possibilities, even if, for PDGs [15], it implies a more expensive processing cost.

Beyond the comparison of costs, a key-point of similarity detection tools relies on their scalability. Indeed, the search for common parts of code could be performed inside a huge project (e.g. for refactoring), between two large projects (e.g. for plagiarism litigation), between an important number of projects, or against a database of projects and could concern a large amount of data. This scalability requires not only practical space and time complexities but also the possibility to incrementally store and re-use the processed data. We also expect from a similarity detection tool to retrieve direct clone clusters rather than numerous simple clone pairs that would later be costly to group by a clustering algorithm. Through subtree fingerprinting, using ASTs to retrieve clones appears as a good compromise between syntactic understanding and scalability.

The usual process for clone detection consists in a dedicated process that looks for approximate matches that might yield a large amount of false-positives. Matches have to be further confirmed or discarded. If there is a change in the set of parameters defining an acceptable degree of similarity, this costly process has to be completely redone. Rather, our approach consists in providing an efficient access to exact AST clones (w.r.t representation abstraction). Instead of providing a global solution, clones are available to be extended into larger approximate matches through various analysis and transformation plugins relying on ASTs.

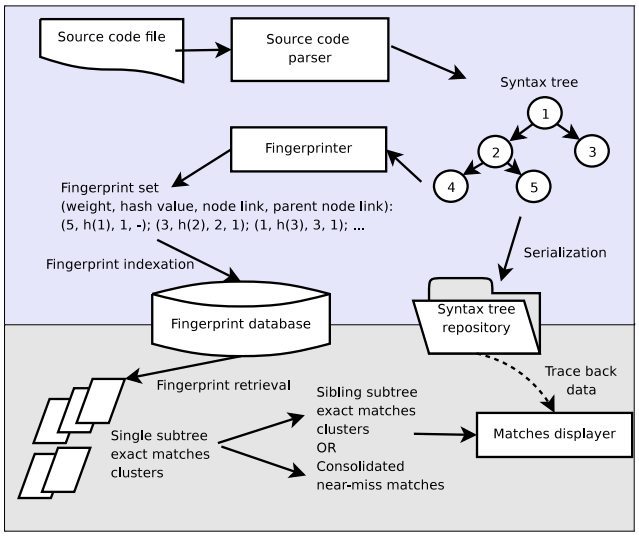

Figure 1. Overview of the system

In this paper, we present in section 2 our system of source code matching that allows large-scale software systems to be stored and processed. It first parses each compilation unit and serializes, stores and references its AST in a repository. Next, each node of this AST is associated with a fingerprint based on a hash value (incrementally computed) of the subtree rooted at the node in question allowing parent and children node information to be reached. These fingerprints are indexed and the system permits the retrieval of clusters of exact matches in the database but also looks for clones matching a (part of a) given single query tree. In both cases, the ASTs of detected clones could be retrieved from the repository to discard the possible false-positives but, contrary to existing methods [2, 11] that artificially choose *bad* hash function to associate a same value for near-miss clones, we prefer syntax tree hashing that avoids collisions, saving us the costly detection of false-positives. Then, for a concrete case study, section 3 shows some results of our system for the Java API source code.

2 Code matching process overview

2.1 Fingerprinting/indexing syntax trees

The main steps of our source code matches retrieval system are summarized in figure 1. Each compilation unit is first parsed into an AST that is lazily serialized and stored, allowing us to fetch backtracing data for a given node without retrieving the complete rooted subtree. Each subtree[1], from these syntax trees obtained from parsing, is represented with a digest form (a fingerprint). The fingerprint of subtree t is a tuple including its weight $w(t)$ (in fact the size of the subtree), its hash value $\mathcal{H}(t)$ reflecting its structural properties computed thanks to a hash function (cf. section 2.3) and pointers to its related subtree root node and to its parent node. A double index is maintained on the fingerprint database: fingerprints are first sorted according by decreasing weight, then by hash value, and also by parent linked node. We implement these indexes using a B+-k-tree [3] (k to $2k$ arity tree indexing structure adapted for mass storage). Thanks to this structure, it is possible to iterate over the database to retrieve first the most weighted clones and to get all of the fingerprints of children subtrees of a given node. Assuming a linear parsing algorithm and an incremental hashing function for the subtrees (see section 2.3) the time complexity is limited by the fingerprint indexing process requiring $O(n \log_k n)$ I/O accesses for a B+-k-tree to index a tree of size n. We study and evaluate several tree fingerprinting methodologies in section 2.3.

2.2 Retrieving clusters of exact matches

2.2.1 Single subtree exact match clusters

A simple iteration over the fingerprint database[2] enables the retrieval of clusters of exact subtree matches: clusters share the same weight and hash value. However, even if false-positives are relatively improbable (syntactically different subtrees with same weight and hash value), they must not be ignored. The length of hash values can be increased to reduce the occurrences of false-positives with a detrimental burden on the size of the database.

To discard false-positives in clusters, we examine a fixed number of children fingerprints in a breadth-first exploration, that drastically reduces the collision probability. For instance, considering two subtrees, rooted at nodes α and β with respective children $a_1, \ldots, a_i$ and $b_1, \ldots, b_i$. If α and β share the same weight and hash values, we compare the weight and hash values of the children pairs $(a_1, b_1), \ldots, (a_i, b_i)$. This is easily done through the parent node pointer of the fingerprint that allows, given a node, the retrieval of the fingerprints of its children. This probabilistic approach of false-positive detection does not require the deserialisation of the syntax trees.

Furthermore, a subtree exact matches cluster C of a given weight implies the existence of clusters of smaller weights containing each subtrees of the trees in C. Iterating from greatest weight to smallest weight in the database allows us to discard these sub-matches.

[1] Since we are not interested in finding tiny clones and we want to minimize the size of our database, only fingerprints whose linked subtrees are greater than a weight threshold are kept.

[2] Rather than iterating over all the database, only fingerprints from a given set of hash values may be fetched if we are interested in clones existing between a given tree and the forest of fingerprinted trees in the database.

2.2.2 Multi-subtrees exact matches clusters

The fingerprint database structure does not address the problem of finding clusters of sequence of consecutive sibling subtrees. Basically, this would require fingerprinting all factors of sequences of child nodes and would consequently multiply the size of the database by a factor up to the maximal arity of the tree. To efficiently cope with this problem, we use a suffix tree or a suffix array to index all of the suffixes (and implicitly the factors) of the sequences of child node fingerprints. This approach has already been used on token sequences or tokenized representations of syntax trees (obtained from complete tree depth traversal) [22, 14], but not yet on sequences of fingerprints representing sibling subtrees, each subtree already being a member of a clone cluster. This allows to minimize the size of handled structures and to syntactically bound considered matches.

Given single subtree exact match clusters (cf. section 2.2.1), our algorithm uses a suffix array to compute consecutive multi-subtrees exact matches. First, an array of all of the single subtree nodes contained in an exact match cluster (i.e. subtrees occurring at least twice) is sorted according to the syntax tree links, such that fingerprints of sibling nodes in the syntax tree are adjacent in the sorted array. Next, all of the segments of consecutive sibling subtrees (represented by their cluster, i.e. their equivalence class identifier) are collected to build a generalized suffix array. This suffix array contains all of the suffixes of fingerprint segments sorted such that matching consecutive sibling subtrees are adjacent in the suffix array. We next create an interval tree from the suffix array: this interval tree is homomorph with the generalized suffix tree (without its leaves) that may be built from the fingerprint segments.

For instance, let us consider the fingerprint segments $s_1 = abcdef$, $s_2 = gbcdeh$ and $s_3 = ijcdeh$: s_1 represents the fact that a, b, c, d, e and f are the fingerprints of six consecutive sibling children of a given node. In the same way as s_2 and s_3 represent other sequences with some shared fingerprint values. We then compute the suffix array and interval tree presented on figure 2 that specifies, besides the parenthood relationship, the suffix links between intervals. Each interval with a lcp (length of common prefix) greater than 1 is suffix-linked to an interval that contains at least the suffixes of the segments of length $\mathrm{lcp} - 1$. If it contains no other segment suffix, this interval is not interesting for clone cluster reporting, since the clones of this interval are included into an interval of greater lcp. The discarded intervals are shown in gray in figure 2; for instance, the interval $\mathrm{de}[7..9]$ only contains 3 suffixes that directly come from interval $\mathrm{cde}[4..6]$. On the other hand, if the suffix interval contains one or several new segments, it has to be considered as a clone cluster; for instance, the interval $\mathrm{cde}[4..6]$ contains not only suffixes of the interval $\mathrm{bcde}[2..3]$, but also of the suffix $s_3[3..5]$. Finally, all of the intervals, minus the discarded ones due to suffix inclusion, represent the clone clusters containing segments of consecutive sibling subtrees.

Building the suffix array can be achieved in a time linear with the cumulated lengths of segments [12] whereas the interval tree construction is also obtained in linear time iterating over the computed lcp-table (also in linear time [13]) using a stack.

2.3 A glimpse on subtree hashing

Avoiding collisions and thus subtree false-positive reports is an essential criterion for the choice of an hash function. An incremental hash function $\mathcal{H}$ is also highly wanted: given a tree α of kind K with child subtrees $a_1, \cdots, a_n$, a hash combination function f must exist such as $\mathcal{H}(\alpha) = f\left(K, (\mathcal{H}(a_1), \cdots, \mathcal{H}(a_n))\right)$ may be computed[3] in constant time. Then the subtrees can be bottom-up fingerprinted. A first requirement is the definition of an assigned hash value (e.g. a randomly generated integer) for each kind of node according to the wished abstraction level. For example one can abstract all the primitive types (*integer*, *long*, *short*, ...) or loop nodes with a single hash value. We studied in [7] several hash combination functions through the quantification of generated collisions on a randomly generated set of trees. These functions were based on cryptographic hash functions (the hash value of a tree is computed by the SHA-1 or MD5 digest of the concatenation of the hash values of its child subtrees) or Karp-Rabin hashing (a polynomial function of the hash values of the child subtrees). Experimental results show that they are nearly equivalent in terms of encountered collisions. For comparison purpose, we also introduced a *bad* hash combination function that sums all the node kind hash values in a subtree (that behaves like metrics).

3 A case study: retrieving clones on the OpenJDK

In order to test several abstraction levels we apply our clone detection tool on the Java classes of the source code of the Open JDK 1.7[4] with the following fingerprinting methodologies.

Standard fingerprinting (`std`): the syntax trees are abstracted regarding to the identifiers with the introduction of commutative operators (*TypeDeclaration* for class

[3] We can derive from the combination function f a commutative function f_c by sorting the hash values of children: f_c may be used for nodes whose child subtrees are commutative (operands of some infix operators, members of a class, ...).

[4] Open JDK 1.7 build 42 of 2008-12-19 with ∼2.5M LOC in 7378 files parsed in 29.9 million nodes by the Eclipse JDT parser

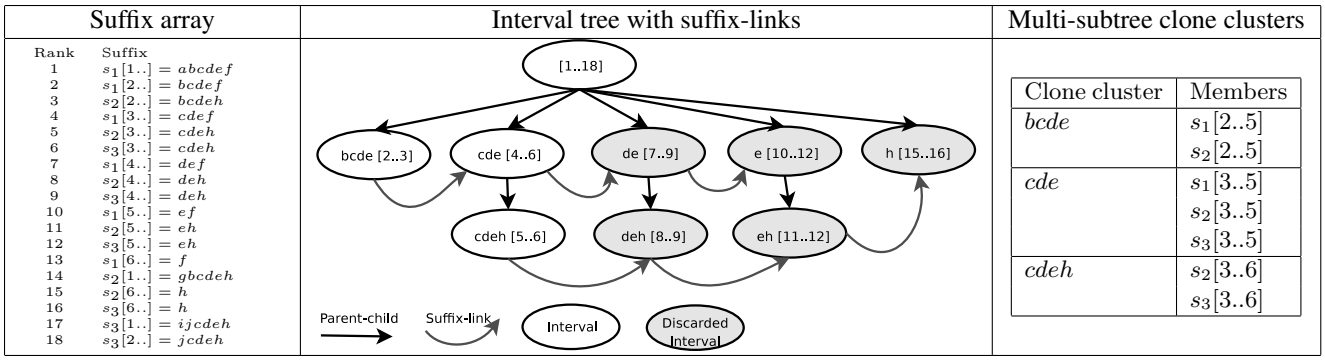

Figure 2. An example of extension of single subtree matches to consecutive sibling nodes between sequences of fingerprints $s_1 = abcdef$, $s_2 = gbcdeh$ and $s_3 = ijcdeh$

members and *InfixOperation*). Only fingerprints for subtrees greater than a set weight threshold w (in terms of number of nodes) are kept. **Fingerprinting with subtree abstraction (`subAbstr`)**: a new level of abstraction is introduced by attributing to all of the subtrees of strictly smaller weight than w' ($w' \leq w$) the same hash value. Even if close to Asta's approach [9] that abstracts subtrees at a depth greater than a given threshold it may detect less false-positives in depth-unbalanced trees. Minor edit operations done on small subtrees are ignored by the fingerprinting process and near-miss clones can be detected with a loss of precision. **Fingerprinting with primitive type abstraction (`primAbstr`)**: since Java does not provide a template system for primitive types, large amount of code may be duplicated for different primitive types. Abstracting primitive types enables the localizing of these chunks of code even if they cannot be factorized. **Fingerprinting using the sum hash function (`sum`)**: the sum hash function does not consider the structure and may induce numerous false-positives on semantically different subtrees sharing the same number of nodes of each type.

The quantitative results obtained[5] with these different methodologies applied on the Open JDK 1.7 can be found in figure 3. Since the notion of *good* matches is a very subjective one, computerized quantification of accuracy and recall of results is very touchy. However if all of the exact subtree matches (considering the tree abstraction level induced by the fingerprinting methodology) can be retrieved, no guarantee can be provided on their semantic value. According to Roy and Cordy's taxonomy of editions scenarios [20], our tree fingerprinting technique is insensitive to formatting

Methodology	Clusters	Clones	Clone pairs
`std` $w \in [10, 50[$	23426	107967	2447949
`std` $w \in [50, 100[$	1390	3637	10511
`std` $w \geq 100$	485	1454	10947
`subAbstr` $w \in [10, 50[$	19344	128331	6908092
`subAbstr` $w \in [50, 100[$	1499	3914	10948
`subAbstr` $w \geq 100$	542	1584	11604
`primAbstr` $w \in [10, 50[$	22892	108601	2560195
`primAbstr` $w \in [50, 100[$	1417	3717	10706
`primAbstr` $w \geq 100$	516	1530	11087
`sum` $w \in [10, 50[$	23953	115914	2651345
`sum` $w \in [50, 100[$	1446	3767	10623
`sum` $w \geq 100$	522	1529	10987

Figure 3. Clone enumerations for different fingerprint methodologies

change and identifier renaming. If it cannot directly deal with insertion or deletion of statements, the `subAbstr` abstraction can manage minor modification edits and `sum` abstraction code transposition.

We note that the fingerprinting methodologies are more or less neutral in terms of reported clone numbers whereas the size of these clusters may vary especially for small to medium clones when several clusters for a weakly abstracted methodology can be merged by a higher abstracted one. The detection of large clones ($w \geq 100$) is hardly impacted by the various methodologies.

We also studied the localization of clone pairs among the packages: most clones (more than 80%) can be found inside the same package. In terms of source code coverage the clones found using the different fingerprinting methodologies represent less than 8% of the source code (6.0%, 6.1%, 6.4% and 7.4% resp. for `std`, `primAbstr`, `sum` and `subAbstr`). Unsurprisingly the `subAbstr` methodology (with $w' = 5$) induces the better clone code coverage: the additional large clones reported are comparatively rele-

[5]The parsing and fingerprint processes using the four methodologies were executed with the Sun client JVM 1.6 on an Intel P8600 2.4 Ghz with 4 GB of RAM in about 15 minutes. Iterating over the fingerprint database, the creation of exact match clusters and the extension of single exact matches to consecutive sibling exact matches were executed in less than 3 minutes.

vant according to a preliminary analysis on a little sample. Some characteristic clone samples detected with the different methodologies could be found in [7].

4 Conclusion and future works

In this paper we focused on a method using fingerprinting on syntax trees to retrieve clone clusters of exact matches across sets of projects or between a given project and a database of projects. We proposed a technique to aggregate continuous sequences of matched single clones thanks to a suffix array. Then we focused on evaluating several hash functions to minimize collisions between false positives: our results show that both Karp-Rabin hashing and cryptographic hashing could be efficiently used on syntax trees. Beyond these technical consideration, the most important parameter in the fingerprinting process is the level of abstraction for the handled syntax trees: different fingerprinting methodologies linked to different abstraction levels were tested. Finally, we reviewed the cases where an exact matching system, based on syntax tree fingerprinting, showed its limits. However the system described must be seen as a foundation permitting later extension of exact matches to near-miss matches, in order to detect more sophisticated obfuscation patterns, plagiarized code like modifications of control structures or inlining/outlining of functions. Extension methods from exact match germs have already been studied through local alignment on linear token sequences [23] but not on AST structures. The formal definition and description of a scalable extension process on ASTs that could consolidate not only sibling non-consecutive exact matches but also cousin exact matches with a behavior adapted to the types of encountered nodes remain.

References

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*. IEEE CSP, 1995.

[2] 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.

[3] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

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

[5] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *SCAM*, Montreal, October 2002. IEEE-CS.

[6] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection. *IEEE Trans. Information Theory*, jul 2004.

[7] M. Chilowicz, E. Duris, and G. Roussel. Syntax tree fingerprinting: a foundation for source code similarity detection. Technical Report IGM2009-03, Université Paris-Est Marne-la-Vallée, 2009. `http://igm.univ-mlv.fr/LabInfo/rapportsInternes/2009/03.pdf`.

[8] P. Clough. Plagiarism in natural and programming languages: an overview of current tools and technologies. Internal Report CS-00-05, University of Sheffield, 2000.

[9] W. S. Evans, C. W. Fraser, and F. Ma. Clone detection via structural abstraction. In *WCRE '07*, pages 150–159, Washington, DC, USA, 2007. IEEE CS.

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

[11] 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.

[12] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. ICALP*, volume 2719 of *LNCS*, 2003.

[13] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. *Combinatorial Pattern Matching (LNCS)*, pages 181–192, 2001.

[14] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE '06*, pages 253–262, Washington, DC, USA, 2006. IEEE-CS.

[15] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *KDD '06*, pages 872–881, New York, NY, USA, 2006. ACM Press.

[16] T. Mende, F. Beckwermert, R. Koschke, and G. Meier. Supporting the grow-and-prune model in software product lines evolution using clone detection. In *CSMR 2008*, pages 163–172, Athens, Grece, 2008. IEEE.

[17] A. Parker and J. Hamblen. Computer algorithms for plagiarism detection. *IEEE Trans. on Educ.*, 32(2), 1989.

[18] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarism among a set of programs JPlag. *Journal of Univ. Comp. Sci.*, 8(11), Nov. 2002.

[19] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC '08*, pages 172–181, 2008.

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

[21] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In A. P. NY, editor, *Proc. Int. Conf. on Management of Data*, 2003.

[22] R. Tairas and J. Gray. Phoenix-based clone detection using suffix trees. In *ACM-SE 44*, pages 679–684, New York, NY, USA, 2006. ACM.

[23] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *APSEC '02*, page 327, Washington, DC, USA, 2002. IEEE Computer Society.

[24] M. J. Wise. YAP3: improved detection of similarities in computer program and other texts. In *Proc. the Conf. on Comp. Sci. Educ.* SIGCSE, ACM, Feb. 1996.