Syntax tree fingerprinting: a foundation for source code similarity detection

Michel Chilowicz, Etienne Duris and 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

Plagiarism detection and clone refactoring in software depend on one common concern: finding similar source chunks across large repositories. However, since code duplication in software is often the result of copy-paste behaviors, only minor modifications are expected between shared codes. On the contrary, in a plagiarism detection context, edits are more extensive and exact matching strategies show their limits.

Among the three main representations used by source code similarity detection tools, namely the linear token sequences, the Abstract Syntax Tree (AST) and the Program Dependency Graph (PDG), we believe that the AST could efficiently support the program analysis and transformations required for the advanced similarity detection process.

In this paper we present a simple and scalable architecture based on syntax tree 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 [3, 8, 4, 29]. Several tools address the problem of identifying software clones that come from copy-paste modifications sometimes followed by slight modifications [7, 34]. A related problem is the detection of software plagiarism, i.e. copies with intentional obfuscations, for instance for license issues or in evaluation of student projects [31, 11]. 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 [40, 32, 30, 9].

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. Linear representations by (abstracted) token sequences of source code allow the comparison algorithms to be efficient, inspired from text pattern-matching and genomics, but they hide the structural organization of the program. For instance, they might identify clones corresponding to parts of code in between two distinct syntactic blocks, that are often not relevant, e.g. in between functions like `return; } public static void`. From the point of view of similarity detection, complex obfuscation patterns cannot be dealt with without manipulation of a structured representation like the ASTs. Let us consider the simple example where an operator is known to be commutative, like `+` in the expressions $e_1 = \mathtt{a} * \mathtt{b} + \mathtt{c}/\mathtt{d}$ and $e_2 = \mathtt{c}/\mathtt{d} + \mathtt{a} * \mathtt{b}$: a linear sequence of tokens does not enable the perception of such equivalence (unless abstracting all the operators). Apart from this example,

several parameters could be considered for defining an *acceptable* degree of similarity that could rely on language constructs, type abstractions, operator properties or various thresholds. Thus, richer representations, like the ASTs or the PDGs, permit fine-grained analysis, manipulation and comparison possibilities, even if they often imply more expensive processing.

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.

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), like `a * b` and `c/d`, and to their parent nodes, like e_1 and e_2. Instead of providing a global solution, clones are available to be extended into larger approximate matches through various analysis and transformation plugins.

In this paper, after briefly presenting related works in section 2, we present in section 3 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 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. Section 4 presents several studied tree fingerprinting methodologies, and evaluates their theoretical efficiency. For a concrete case study, section 5 shows some results of our system for the Java API source code. Prior to the conclusion, we study in section 6 the behavior of our approach on different clone scenarios.

2 State of the art and issues raised

Bellon *et al.* propose in [7] an extensive survey of clone detection tools. Roy and Cordy give in [34] a short but relevant overview testing existing techniques against several editing scenarios. Thus, instead of an exhaustive presentation of existing tools, we present in this section some characteristics of the four main approaches in order to put our contribution in context.

2.1 Metrics based techniques

Metrics like Halstead measures [17] are widely used in software engineering to quantitatively assess properties on a given project by associating to each chunk of code (package, compilation unit, method, ...) a vector. Unfortunately clone detection techniques based on metrics perform very poorly due to the high sensitivity of most metrics to minor edits and *a fortiori* to plagiarism cases. Thus nowadays few clone detection tools use pure metrics methods. Nevertheless we can quote Deckard [19, 13] that computes degraded hash values for subtrees of an AST applying an LSH-hash function [14] on node type-counting metrics.

2.2 Token sequence based techniques

After simple string matching techniques (easy defeated by simple formatting edits), token based matching is the first historical approach to deal with source code similarity detection using sequence matching algorithms. As a first step, source code projects are tokenized to provide sequences of tokens that are supplied to string matching algorithms, usually inspired from genomics applications. The tokenized representation is generally abstracted (e.g. identifiers are not distinguished). The chosen algorithm depends on the source comparison context and on the computation capabilities. To extensively compare a single pair of projects represented by two sequences of n tokens, their Levenstein distance may be computed and local alignments [18] are inferred with the time complexity of $O(n^2)$ using dynamic programming algorithms. This approach is not scalable either for the comparison of a set of projects (all pairs have to be compared) nor for the search of similarities of a single project against a database of projects. For such contexts, token sequence fingerprinting appears unavoidable. It consists in computing fingerprints of token k-grams that will be inserted in a database ($n-k+1$ fingerprints for n tokens). A fingerprint selection strategy (like Winnowing used by the Moss [30] tool) can be used to divide by a constant the amount of fingerprints kept. The greedy string tiling algorithm [39] implemented by JPlag [20] permits the search of exact matches between two sequences of tokens through increasing k-grams fingerprinting in experimental quasi-linear time, but requires comparison of all of the pairs of projects. Another approach is based on the factorization of the tokenized source code into synthesized functions, allowing for a quantification of similarities at a function-level through call-graph [10].

Our system shares some techniques with these approaches, like suffix arrays for sequence matching [28] or for the incremental fingerprinting processing, like Karp-Rabin [22].

2.3 Tree based techniques

In order to exploit the syntactic properties of the programs, another approach considers the syntax trees of compilation units obtained through parsing. Next, an abstracted version of the trees is considered (e.g. abstraction of identifiers, loop constructs...) to achieve a better recall. Similar to the token sequences, edit distances can be computed for each pair of trees [41], but tree fingerprinting allows for a more scalable behavior. This method is widely used by tools advertising themselves as clone retrieval systems, but strangely not by the plagiarism detection tools. Since comparing nm subtrees of m projects of size n (in terms of nodes) for exact equality detection would require $O((nm)^2)$ comparisons with a naive approach, all subtrees are rather fingerprinted and put in buckets according to their hash value. To retrieve not only exact subtree matches (considering the level of abstraction of the tree), but also near-miss ones, degraded hash functions are used for fingerprinting (like in CloneDr [5] where subtrees below a weight level are hashed with a single value). The drawback of this approach is the potential increase of false-positive clones.

Comparatively, our system focuses on exact tree matching retrieval through the use of a *good* hash function minimizing collisions between false-positives. Quite close to the Koschke *et al.* approach [25] that uses suffix tree clone detection on AST node types, our system uses AST node fingerprints that reflect the whole underlying subtree. This provides an efficient clustering mechanism for exact match of sequences of sibling subtrees. Direct access to indexed ASTs allows further analysis and manipulations to extend neighboring matches into larger near-miss similarities.

2.4 PDG based techniques

Generating PDGs for source code through program slicing is an interesting semantic approach to detect complex syntactic modifications [16, 24, 26, 15]. However, finding isomorphisms between subgraphs is very costly (being a NP-complete problem in general): comparison of large sets of projects implies some compromises. For plagiarism detection, pruning filters may be used to reduce the number of graph pairs to be compared [27].

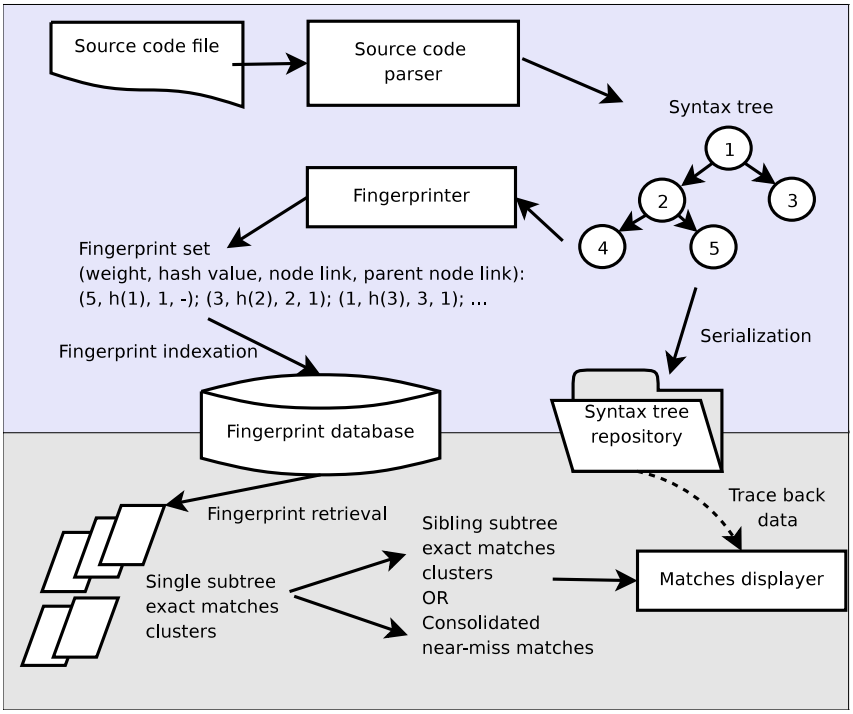


Figure 1: Overview of the system

3 Code matching process overview

We detail in this section the main steps of our source code matches retrieval system, summarized in figure 1.

3.1 Parsing the source

Given a compilation unit, we want to produce a syntax tree representation to perform the fingerprinting process. The knowledge of the token definitions and of the context-free grammar of the language handled permits the generation of a lexer and parser with dedicated tools (lexer generators producing a DFA and parser generators allowing linear time syntactic analysis using for example LALR(k) or LR(k) algorithms). To obtain the abstract syntax tree, either a post-processing of the concrete syntax tree or the integration of customized actions with the grammar specification is needed. Specifying common abstract tree nodes for different languages of the same type (procedural, object, functional...) permits the use of abstract post-processing normalization tools, for instance to remove trivial, useless or unreachable instructions or to allow cross-language plagiarism detection. Once computed, the abstract syntax tree is stored in a repository for further references, e.g. for false-positive checking or match back-tracing during the result reporting. The tree is saved level by level in a customized binary XML representation with a companion index allowing the user to randomly access the nodes and a lazy deserialization: when a node is requested for deserialization its heirs are not automatically deserialized but are done so *on demand*.

3.2 Fingerprinting and indexing syntax trees

Each subtree, from the 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 4) and a pointer $p(t)$ to its related subtree root node t 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 [6] (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 (allowing computation of the hash value of a subtree with the knowledge of hash values of its child subtrees in $O(1)$) 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 4.

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. Another solution can also be proposed to aggregate sibling subtrees of sub-threshold weight to represent them with a single composed fingerprint.

In addition, in order to spare storage space by discarding the weight field, the database can be distributed in sub-databases containing only fingerprints of subtrees of a given weight. The fingerprint hash lengths can be adapted for the subtree sizes: intuitively, subtrees of greater weight require hash values of greater length to minimize false-positive collisions (cf. section 4).

3.3 Retrieving clusters of exact matches

3.3.1 Single subtree exact match clusters

A simple iteration over the fingerprint database 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, see section 4.1), 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.

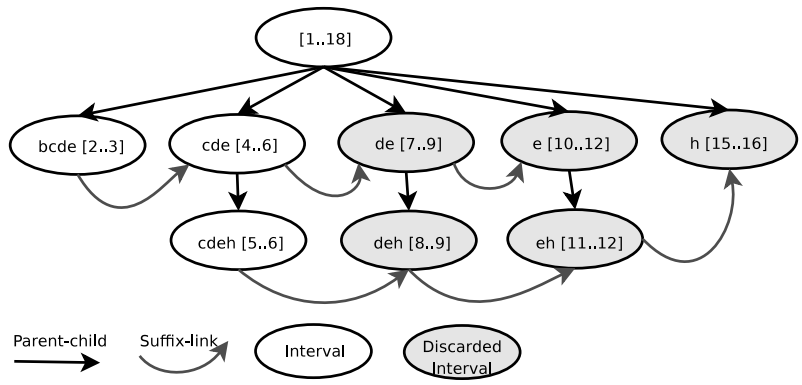
To discard false-positives in clusters, we examine a fixed number of children fingerprints in a breadth-first exploration, that drastically reduces the collision probability. For instance, considering two subtrees, rooted at nodes α and β with respective children $a_1, \ldots, a_i$ and $b_1, \ldots, b_i$. 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, an 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.

3.3.2 Multi-subtrees exact matches clusters

The fingerprint database structure does not address the problem of finding clusters of sequence of consecutive sibling subtrees (except for small trees through composed fingerprints, cf. 3.2). 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) [36], or on AST node types [25], but not yet on sequences of fingerprints representing sibling subtrees. This allows minimizing the size of handled structures and to syntactically bind matches, contrary to pure token sequences used (e.g. a match cannot contain a large function and stop with the duplicated first statement of the next function: such a match must be split).

Suffix array		Interval tree with suffix-links
Rank	Suffix	
1	$s_1[1..] = abcdef$	
2	$s_1[2..] = bcdef$	
3	$s_2[2..] = bcdeh$	
4	$s_1[3..] = cdef$	
5	$s_2[3..] = cdeh$	
6	$s_3[3..] = cdeh$	
7	$s_1[4..] = def$	
8	$s_2[4..] = deh$	
9	$s_3[4..] = deh$	
10	$s_1[5..] = ef$	
11	$s_2[5..] = eh$	
12	$s_3[5..] = eh$	
13	$s_1[6..] = f$	
14	$s_2[1..] = gbcdeh$	
15	$s_2[6..] = h$	
16	$s_3[6..] = h$	
17	$s_3[1..] = ijcdeh$	
18	$s_3[2..] = jcdeh$	



Clone cluster	*bcde*	*cde*	*cdeh*
Members	$\{s_1[2..5], s_2[2..5]\}$	$\{s_1[3..5], s_2[3..5], s_3[3..5]\}$	$\{s_2[3..6], s_3[3..6]\}$

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 $s_3 = ijcdeh$

Given single subtree exact match clusters (cf. section 3.3.1), 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. a, f, g, i and j are fingerprints expressed once in s_1, s_2 and s_3 but they are present in other segments not explicited here. 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 $\text{lcp} - 1$. 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 de[7..9] only contains 3 suffixes that directly come from interval 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 cde[4..6] contains not only suffixes of the interval 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 [21] whereas the interval tree construction is also obtained in linear time iterating over the computed lcp-table (also in linear time [23]) using a stack.

3.4 Finding exact single node matches between a query tree and a database

Given a single source code project, we also want to search for existing exact matches between this project and all of the projects in the database. This problem can be seen as a specific case of those described in 3.3: instead of iterating on all of the database, we limit the exploration to fingerprint matching subtrees of the query tree. Initially, the fingerprints of the nodes of the query tree are computed[1]. Starting form the root node, and in breadth-first order, these fingerprints are then searched in the database to retrieve clusters of single node clones As in 3.3, sub-matches can be discarded and multi-node exact match clusters (clusters of exact matches of sibling nodes) can be retrieved. If a match cluster is found between a subtree c_q of the query tree and subtrees from other trees in the database and we are not interested in matches containing subtrees under c_q, we can stop the search for subtrees under c_q. Thus it enables the detection of exact clones of compilation units through a single database query.

4 Tree fingerprinting methodologies

Given a tree (AST), with internal nodes and leaves, we want to index by a digest value each subtree of this tree, using a hash function. The first problem encountered is to find such a hash function, with the *good* properties. Since we want to represent the syntax tree and not only its leaves token sequence, we have to use a hash function that is sensitive to the tree structure: thus, we must take into account the internal nodes of the AST to compute the hash value of a subtree. Furthermore, since we want our process to be scalable, and efficient for a large amount of data, it would be better to use an incremental hash function, that only needs the hash values of its direct children to compute the hash value of a subtree. This means that hashing all of the subtrees of a given AST could be made in linear time.

We split the hash function problem in two parts. The first part concerns the study of a hash function for an individual node (without subtree): how to represent the syntactic properties of a single node with an integer? The second part has to do with assembling hash values of individual nodes to infer hash values for subtrees (collection of nodes with sibling and parent relations): we propose and study three hash functions. Finally, we evaluate the theoretical effectiveness of these fingerprinting techniques.

4.1 Hash functions and node-type vector

We characterize each subtree of the syntax tree of a compilation unit via a *good* hash function. A hash function is considered *good* if it minimizes the probability p of hash value collision for two structurally different subtrees. The choice of the length 2^k of the hash value depends on the cardinality of handled tree space that induces the probability of collision p (bound by $\frac{1}{2^k}$ for an infinite set of tree).

We define the node-type vector as the vector specifying for each single node-type (variables, constants, statements, block statements, conditional structures, loop structures...) a constant hash value: this vector is unique for all hash computation for a given language and abstraction level. Depending on the desired abstraction level, certain types of nodes can be represented by the same hash value. For example, it is possible to abstract all of the loop structures (*while*, *for*, *repeat...until*) with a unique hash value since common obfuscation patterns use simple transformation of loop structures. Another concern is related to the abstraction of primitive types. Obfuscation could easily replace a boolean or a byte type by another embracing primitive type such as an integer: it could then be pertinent to use the same hash value for some primitive types or, more generally, for numerical values.

On the contrary, for copy-paste exact match detection, it might be useful to include additional parameters for the hash value computation of an abstract node (such as the name of an identifier).

[1] If the query tree is already present in the database, this step can be skipped.

Finally choosing the better abstraction for hash values of individual nodes is a compromise depending on the language and the application: since obfuscation patterns are expected, maximizing the abstraction level can be considered at the cost of a loss of precision. We then consider a unique mapping between the space of individual nodes and integers, two individual nodes with the same hash value being later considered as an exact match.

A practical solution adopted for our system consisted in setting a random integer value to each type of node, through the cryptographic hash (with MD5 for example) of a char-sequence representing the node type in our AST. For a node a, we note this node hash value $V(a)$. For the set A of (abstracted) node types for a language we obtain the node type hash vector: $\mathcal{V}(A) = \big[V(A_1), V(A_2), \cdots, V(A_{|A|})\big]$.

4.2 Candidate hash functions

We propose and examine the advantages and weaknesses of some candidate hash functions to be used for the fingerprinting process of the syntax tree.

Sum hashing (node type metrics) Simple metrics for subtrees consist in counting each type of node in them: then, each subtree is characterized by a vector where the i-th component is the number of nodes of type i. This approach has been successfully used by Deckard [19] to hash syntax trees: the principle is extended from counting single node types to counting small subtrees and then a LSH-hash function is applied on each count-vector to clusterize them according to their proximity following their Euclidean distance. Since this method ignores the structural organization of trees, near-miss and even completely irrelevant matches can be retrieved. For our tests, presented in section 4.3, we introduce the sum hash function that is the product of the node type-count vector by the node type hash vector (each node type is defined by a randomly-set integer).

Dyck word hashing As previously seen, a subtree can be mapped to a representative integer. Here, we propose to represent a subtree as a Dyck word, i.e. a serialized form of the subtree obtained from depth-traversal. Contrary to a node type-count vector, a Dyck word takes into account the structural properties of the tree.

The Dyck word $\mathcal{D}(\alpha)$ for a leaf α is defined as the integer sequence $V_{\triangleleft}(\alpha)V_{\triangleright}(\alpha)$ whereas the Dyck word $\mathcal{D}(\beta)$ of an internal node β with a children set $\{b_1, \cdots, b_k\}$ is recursively defined as the concatenation $V_{\triangleleft}(\beta) \cdot \mathcal{D}(b_1) \cdots \mathcal{D}(b_k) \cdot V_{\triangleright}(\beta)$. We compute the values $V_{\triangleleft}(\eta)$ and $V_{\triangleright}(\eta)$ for a node η from the node type vector $V(\eta)$ using for example a cryptographic hash function[2].

Then, we build a Karp-Rabin hash function $\mathcal{K}$ on the Dyck word. A Karp-Rabin hash function [22] can be incrementally computed. For our purpose we want to efficiently compute the hash function of a subtree given its root and its children subtrees. Indeed, the Karp-Rabin hash function $\mathcal{K}$ is a polynomial function on a word of integers $m = a_1 a_2 \cdots a_{|m|}$ defined as follows, with the base B being chosen as a prime number (e.g. 33, 65599...) in order to limit hash collisions:

$$\mathcal{K}(m) = (\ldots((a_1 \times B + a_2) \times B + a_3)\ldots) \times B + a_{|m|}$$

We deduce a formula to recursively compute the hash value of concatenated subsequences of integers $m = s_1 s_2 \cdots s_n$ knowing the hash value and the length of each subsequence s_i:

$$\mathcal{K}(m) = \Big(\cdots\Big(\mathcal{K}(s_1) \times B^{|s_2|} + \mathcal{K}(s_2)\Big)\cdots\Big) \times B^{|s_n|} + \mathcal{K}(s_n)$$

Thus $\mathcal{K}(m)$ can be computed in time $t(m) = \sum_{1 \leq i \leq n} (\log_2(|s_i|) + t(s_i))$ using a fast exponentiation. The computation of the hash values of all nodes of a tree of size N can be made in $O(N \log N)$.

[2]The use of a polynomial formula to infer these values (e.g. $V_{\triangleleft}(\eta) = B \cdot V(\eta)$ where B is a prime number) should be avoided because it may degrade the onto property of the hash function.

Hash length	KR ($B = 32$)	KR ($B = 33$)	MD5	SHA-1	Perfect hashing
64 bits	1.698970	$-\infty$	$-\infty$	$-\infty$	-7.567030
40 bits	5.381287	$-\infty$	$-\infty$	$-\infty$	-0.341989
39 bits	5.418374	0	0	0	-0.041436
32 bits	6.321192	2.041393	2.037426	2.037426	2.064458
24 bits	2.426876	4.473764	4.474012	4.473385	4.474245
16 bits	8.878016	6.882354	6.882908	6.882375	6.882490
8 bits	10.175532	9.290719	9.290716	9.290721	9.290730

Figure 3: Hash pair collisions ($\log_{10}(c)$) on 10^6 127-weighted trees

Cryptographic hashing Finally we consider indexing nodes of the tree by a cryptographic hash function such as SHA-1 [35] or MD5 [33]. Thus, given a tree, each node is hashed bottom-up: leaves are indexed first (as described in section 4.1), and the hash value of a node is the cryptographic hash of the concatenation of the node hash with the hash values of the children subtrees. This provides us with a completely incremental indexation process, linear in the number of nodes in the tree. This hash method is expected to be a good approximation of a perfect hash function.

Given a cryptographic hashing function $\mathcal{C}$, such as SHA-1, we define the hash function $\mathcal{H}_\mathcal{C}$. It is defined recursively for a tree t rooted at node r and with children subtrees $t_1 \cdots t_n$ as follows:

$$\mathcal{H}_\mathcal{C}(t) = \mathcal{C}(V(r) \cdot \mathcal{H}_\mathcal{C}(t_1) \cdot \mathcal{H}_\mathcal{C}(t_2) \cdot ... \cdot \mathcal{H}_\mathcal{C}(t_n))$$

The size of hash values generated by cryptographic functions is reduced by selecting first or last bytes with a conventional order (e.g. big-endian).

Exploiting the commutativity of nodes Since some types of node from the grammar of a language can be defined as commutative (e.g. the order of operands of the addition operator can be ignored) structurally-sensitive hash functions like Karp-Rabin and cryptographic functions can be adapted: the concatenated child subtrees hash values can be sorted by increasing hash values prior to their combination.

4.3 Practical quantification of hash function efficiency

In order to test the efficiency and speed of the previously defined hash functions, we compute hash values on sets of randomly generated binary trees of different sizes. The collision results are expressed in figure 3 for randomly generated binary trees of 127 nodes (and height of 7) and are compared with the expected results of a perfect node types and structurally comprehensive hash function. For each test, we fingerprinted 10^6 random binary trees using 10 different types of internal nodes and 10 types of leaves. We notice in figure 3 that Karp-Rabin and cryptographic hash functions are almost as efficient for limiting false-positive collisions and offer results very close to a theoretically perfect hash function[3].

Even if more simply defined than a non-linear cryptographic hash function, the Karp-Rabin hashing function induces a theoretical detrimental run-time cost on large trees due to the base exponentiation cost (in $O(\log_2 n)$ for a tree of n nodes). However on small trees encountered when parsing source code, MD5 and SHA-1 hashing are experimentally more time costly (respectively 26.3% and 35.0% more costly than sum hashing on trees of 1023 nodes) than Karp-Rabin hashing function (22.8% more costly than sum hashing). Since in practice I/O database time cost is much higher ($\sim$10 times) the choice of the hash function should not rely on this criterion.

Deviation from a theoretical hash function highlighted on small trees hashing when the tree space cardinality is smaller than the hash value space cardinality: e.g. $5 \cdot 10^7$ 7-weighted trees can

[3]Concerning Karp-Rabin hashing, we underline the necessity of using a prime base B: for example the use of the base 32 generates numerous hash value collisions.

be enumerated whereas the first collision is encountered (for SHA-1 hashing) for a 41-bit hash (vs. 25-bit hash for a perfect hash function). Thus rather than using a hash value for small subtrees an interesting idea would be to introduce a specific companion repository dedicated to small subtree storage replacing the hash value in the fingerprint database by a link to the subtree.

5 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 [2] (build 42 of 2008-12-19 with ∼2.5M LOC in 7378 files parsed in 29.9 million nodes). We fingerprint the subtrees extracted from syntax trees generated by the Eclipse parser (built with LALR Parser Generator [1]). We propose the following fingerprinting methodologies:

1. Standard fingerprinting (`std`). The syntax trees are abstracted regarding to the identifiers. Only fingerprints for subtrees greater than a set weight threshold w (in terms of number of nodes) are kept.

2. 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 [12] 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 risk of increasing false-positives.

3. Fingerprinting with consideration of commutative operators (`comm`). Subtrees that are a child of a commutative operator are considered as interchangeable and their order is normalized for hash value computation (see 4.2). For our tests on Java source code we considered the *TypeDeclaration* node as a commutative operator (order of field and method declarations in a class is not important) and all *InfixOperation* nodes (even if not semantically correct for all infix operators).

4. 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.

5. Fingerprinting with total node abstraction (`struct`). Only the structure of the syntax trees is considered for the hashing process, i.e. the uniform node type vector.

6. 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[4] with these different methodologies applied on the Open JDK 1.7 can be found in figure 4. 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. We note that `struct` fingerprinting induces almost two times more clone pairs than `sum` fingerprinting: the probability of chunks of code with shared structure being semantically different is high whereas considering only the types of node provides a relatively better accuracy. We observe that the fingerprinting methodologies are more or less neutral in terms of clone numbers whereas the size

[4]The parsing and fingerprint processes using the six methodologies were executed with the Sun client JVM 1.6 on an Intel P8600 2.4 Ghz with 4 GB of RAM in about 24 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.

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` $w' = 5, w \in [10, 50[$	19344	128331	6908092
`subAbstr` $w' = 5, w \in [50, 100[$	1499	3914	10948
`subAbstr` $w' = 5, w \geq 100$	542	1584	11604
`comm` $w \in [10, 50[$	23426	107967	2447949
`comm` $w \in [50, 100[$	1390	3637	10511
`comm` $w \geq 100$	485	1454	10947
`primAbstr` $w \in [10, 50[$	22892	108601	2560195
`primAbstr` $w \in [50, 100[$	1417	3717	10706
`primAbstr` $w \geq 100$	516	1530	11087
`struct` $w \in [10, 50[$	20050	126169	6670865
`struct` $w \in [50, 100[$	1478	3861	10910
`struct` $w \geq 100$	534	1571	11603
`sum` $w \in [10, 50[$	23953	115914	2651345
`sum` $w \in [50, 100[$	1446	3767	10623
`sum` $w \geq 100$	522	1529	10987

Figure 4: Clone enumerations for different fingerprint methodologies

Distance	Clone pairs ratio	Average weight
0 (same compilation unit)	7.48%	380
1 (same package)	73.8%	469
2	6.71%	248
3	0.411%	237
4	10.9%	468
$5 - 10$	0.703%	249

Figure 5: Distance of 10947 largest clone pairs ($w \geq 100$) across the code (using `std`)

of these clusters may vary especially for small to medium clones. The detection of large clones ($w \geq 100$) is hardly impacted.

We also studied the localization of clone pairs among the packages: figure 5 shows that most clones 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 (see figure 6). Unsurprisingly `subAbstr` methodology induces the better clone code coverage: the additional large clones reported are comparatively relevant according to a preliminary analysis on a little sample. Finally figure 7 shows the number of clusters and their average size by weight: as expected the extension of sibling exact subtrees allows clones of higher weight to be reported.

6 Scenario-based behavior

In [34], C.K. Roy and J.R. Cordy propose a taxonomy of editing scenarios that is used to compare clone detection techniques. In this section, we will use this taxonomy to present the result of our syntax tree fingerprinting technique and to show how we plan to extend it.

The first set of copy-paste scenarios proposed in [34] concerns changes in whitespaces, comments and formatting. Since our tree fingerprinting technique is based on a complete tree abstraction, these clones are detected regardless of the modifications.

The second set of copy-paste scenarios concerns identifier renaming, parameter permutation in function calls, changes in types and replacement of parameters by expressions. No matter what

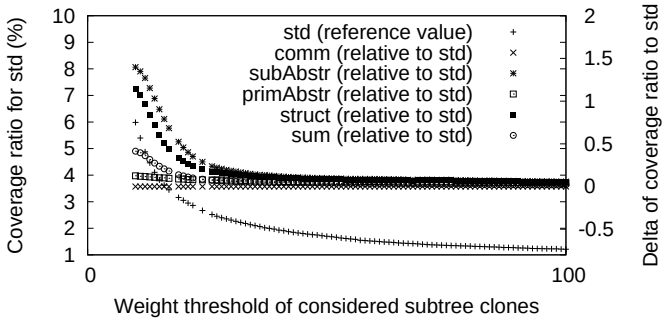


Figure 6: Clone coverage (ratio of duplicated clones weight sum on global syntax tree weight)

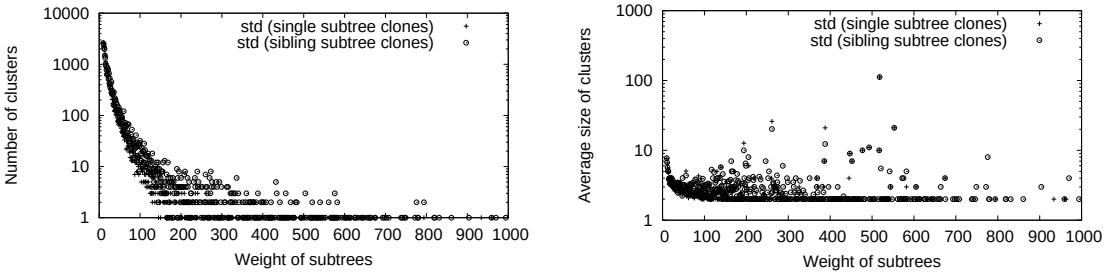


Figure 7: Clone clusters number and average size by weight

abstraction is used, the tree fingerprint is not sensible to identifier renaming. On the contrary, depending on the abstraction used for function calls, types and constants, the tree fingerprint may be affected by the three other types of modifications. Changes in types are not perceptible if `primAbstr` abstraction is used. Clones containing permutations of arguments or changes in expressions may be detected if `subAbstr` is used.

The third set of copy-paste scenarios concerns changes in arguments, insertion or deletion of a line and modification of a line. Clones containing changes in arguments may be detected by `subAbstr`, but the other types of modifications cannot be detected, changing the abstraction. In order to find these modifications, the subtree fingerprint has to be extended to find near miss clones. For instance, using multi-subtrees exact match clusters (see section 3.3), we can detect that two distinct nodes have similar children with the exception of the one inserted, modified or deleted. Thus, a simple post-processing after exact clone matching can enable the detection of these clones.

The last set of scenarios concerns line permutations and a *for*-into-*while* translation. The line permutations can be detected by the same post-processing as in the previous scenarios. The *for*-into-*while* translation detection requires, on top of post-processing, the common abstraction for loop instructions.

This presentation demonstatres that the use of an exact matching technique followed by a degree of post-processing in a specific location of the code may detect the same clones as those detected by techniques based on near-match hashing techniques but at the same time reduces in the first step the false positives. Other post-processings may by applied to detect intentional obfuscations.

7 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 on Dyck words of serialized subtrees 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 under consideration of the syntax trees handled: we tested several fingerprint methodologies, based on different abstraction levels. 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 token sequences [37, 38] 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] LALR parser generator. `http://sourceforge.net/projects/lpg/`.

[2] Open JDK 1.7. `https://jdk7.dev.java.net/`.

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

[4] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms®: Program transformations for practical scalable software evolution. In *ICSE*. IEEE-CS, 2004.

[5] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98*, page 368, Washington, DC, USA, 1998. IEEE-CS.

[6] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[7] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Software Eng.*, 33(9):577–591, 2007.

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

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

[10] Michel Chilowicz, Étienne Duris, and Gilles Roussel. Finding similarities in source code through factorization. In *LDTA'08*, ENTCS, 2008.

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

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

[13] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ICSE '08*, pages 321–330, New York, NY, USA, 2008. ACM.

[14] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB '99*, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[15] Susan Horwitz and Thomas W. Reps. The use of program dependence graphs in software engineering. In *ICSE*, Melbourne, 1992. ACM Press.

[16] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Lang. and Sys.*, 12(1), 1990.

[17] Halstead Maurice Howard. *Elements of Software Science*. Elsevier, New York, 1977.

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

[19] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07*, pages 96–105, Washington, DC, USA, 2007. IEEE-CS.

[20] JPlag. `https://www.ipd.uni-karlsruhe.de/jplag`.

[21] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. ICALP*, volume 2719 of *LNCS*, 2003.

[22] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2), 1987.

[23] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. *Combinatorial Pattern Matching (LNCS)*, pages 181–192, 2001.

[24] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *SAS*, volume 2126 of *LNCS*, Paris, 2001. Springer.

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

[26] Jens Krinke. Identifying similar code with program dependence graphs. In *WCRE*, 2001.

[27] Chao Liu, Chen Chen, Jiawei Han, and Philip 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.

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

[29] Thilo Mende, Felix Beckwermert, Rainer Koschke, and Gerald 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.

[30] Moss. `http://theory.stanford.edu/~aiken/moss`.

[31] Alan Parker and James Hamblen. Computer algorithms for plagiarism detection. *IEEE Trans. on Educ.*, 32(2), 1989.

[32] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarism among a set of programs JPlag. *Journal of Univ. Comp. Sci.*, 8(11), November 2002.

[33] R. Rivest. Md5 (rfc 1321). `http://tools.ietf.org/html/rfc1321`, 1992.

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

[35] Secure Hash Standard. `http://www.itl.nist.gov/fipspubs/fip180-1.htm`, 1995.

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

[37] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On detection of gapped code clones using gap locations. In *APSEC '02*, page 327, Washington, DC, USA, 2002. IEEE Computer Society.

[38] Richard Wettel and Radu Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *SYNASC '05*, page 63, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Michael J. Wise. Running karp-rabin matching and greedy string tiling. Technical Report 463, Dep. of Comp. Sci., Sidney Univ., March 1994.

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

[41] 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.

A Appendix: examples of clones in the OpenJDK

In this appendix, we present some examples (summarized in figure 8) of code clones found on the OpenJDK with our system. We classify these clones according to several criteria. The first criterion concerns the proximity of each component of the clone pair in the source code. As seen in figure 5, most of the clones are either found in the same class or in the same package. Following a code factorization goal, two clones in a same class may be merged into a new function in the same class; if the clones are far apart, the creation of a new helper class containing the common code can be envisaged. We have only studied the proximity in terms of package location and not related to the positioning in the class ancestry tree. In fact some clones found in sibling or near cousin classes could be merged into a common ancestor abstract class. Another useful criterion analyzed concerns the hash methodologies able to identify these clones: thus we provide some examples showing that a higher abstraction level of the syntax tree may increase the recall of matches but may decrease the accuracy as in some false-positives presented . Finally we note, as a third criterion, the usefulness of the clone report for refactoring. In fact, the largest clones found in the OpenJDK are automatically generated code and their refactoring may not be pertinent. Other clones found with the `primAbstr` methodology highlight repetitive code adapted to different primitive types that cannot be merged. Only a small part of the source code can actually be refactored, but sometimes with a detrimental effect on runtime costs.

Clone figure	Weight	Package distance	Detected with	Refactoring usefulness
9	115	0	`subAbstr`, `primAbstr`, `struct`, `sum`	Moderate (symetric code)
10	50	0	`sum`	Low (hand-coded Java lexer)
11	55	3	`struct`	None (false positive)
12	63	2	`subAbstr`, `struct`	High (standard code)
13	333	0	`subAbstr`, `primAbstr`, `struct`, `sum`	Low in Java (sort code for different primitive types)

Figure 8: Evaluation of sample code clones

java.lang.StringCoding.decode()	java.lang.StringCoding.encode()

static char[]
decode (String charsetName, **byte**[]ba, **int** off, **int** len)
  throws UnsupportedEncodingException
  {
  StringDecoder sd = deref (decoder);
  String csn = (charsetName == **null**)
  ? "ISO−8859−1" : charsetName;
  if ((sd == **null**)
  || !(csn.equals (sd.requestedCharsetName ())
  || csn.equals (sd.charsetName ())))
	static byte[]
	encode (String charsetName, **char**[]ca, **int** off, **int** len)
	  throws UnsupportedEncodingException
	  {
	  StringEncoder se = deref (encoder);
	  String csn = (charsetName == **null**)
	  ? "ISO−8859−1" : charsetName;
	  if ((se == **null**)
	  || !(csn.equals (se.requestedCharsetName ())
	  || csn.equals (se.charsetName ())))

{
  sd = **null**;
  try
  {
  Charset cs = lookupCharset (csn);
  if (cs != **null**)
  sd = **new** StringDecoder (cs, csn);
  }
  catch (IllegalCharsetNameException x)
  {
  }
  if (sd == **null**)
  throw new UnsupportedEncodingException (csn);
  set (decoder, sd);
  }
  return sd.decode (ba, off, len);
  }
	{
	  se = **null**;
	  try
	  {
	  Charset cs = lookupCharset (csn);
	  if (cs != **null**)
	  se = **new** StringEncoder (cs, csn);
	  }
	  catch (IllegalCharsetNameException x)
	  {
	  }
	  if (se == **null**)
	  throw new UnsupportedEncodingException (csn);
	  set (encoder, se);
	  }
	  return se.encode (ca, off, len);
	  }

Figure 9: Example of intra-class clone weighing 115 nodes detected by `subAstr`, `primAbstr`, struct and `sum` but not by `std` and `comm` methods unless the method header is discarded

sun.tools.java.Scanner.xscan()	sun.tools.java.Scanner.xscan()

switch (ch = in.read ())
  {
  case '=':
  ch = in.read ();
  token = ASGRSHIFT;
  return retPos;
 	
  case '>':
  if ((ch = in.read ()) == '=')
  {
  ch = in.read ();
  token = ASGURSHIFT;
  return retPos;
  }
  token = URSHIFT;
  return retPos;
  }
	switch (ch = in.read ())
	  {
	  case '<':
	  if ((ch = in.read ()) == '=')
	  {
	  ch = in.read ();
	  token = ASGLSHIFT;
	  return retPos;
	  }
	  token = LSHIFT;
	  return retPos;
	
	  case '=':
	  ch = in.read ();
	  token = LE;
	  return retPos;
	  }

Figure 10: Intra-method clone weighing 50 nodes in an hand-made Java scanner detected only with `sum`

java.util.zip.ZipEntry	java.beans.FeatureDescriptor

public
ZipEntry (ZipEntry e)
{
  name = e.name;
  time = e.time;
  crc = e.crc;
  size = e.size;
  csize = e.csize;
  method = e.method;
  extra = e.extra;
  comment = e.comment;
}
	FeatureDescriptor (FeatureDescriptor old)
	{
	  expert = old.expert;
	  hidden = old.hidden;
	  preferred = old.preferred;
	  name = old.name;
	  shortDescription = old.shortDescription;
	  displayName = old.displayName;
	  classRef = old.classRef;
	
	  addTable (old.table);
	}

Figure 11: False-positive clone with the same parse tree structure

javax.swing.event.EventListenerList

```
// If so, remove it
if (index != −1)
  {
    Object[]tmp = new Object[listenerList.length − 2];
    // Copy the list up to index
    System.arraycopy (listenerList, 0, tmp, 0, index);
    // Copy from two past the index, up to
    // the end of tmp (which is two elements
    // shorter than the old list)
    if (index < tmp.length)
      System.arraycopy (listenerList, index + 2, tmp, index,
                        tmp.length − index);
    // set the listener array to the new array or null
    listenerList = (tmp.length == 0) ? NULL_ARRAY : tmp;
  }
```

javax.swing.ArrayTable

```
// If so, remove it
if (index != −1)
  {
    Object[]tmp = new Object[array.length − 2];
    // Copy the list up to index
    System.arraycopy (array, 0, tmp, 0, index);
    // Copy from two past the index, up to
    // the end of tmp (which is two elements
    // shorter than the old list)
    if (index < tmp.length)
      System.arraycopy (array, index + 2, tmp, index,
                        tmp.length − index);
    // set the listener array to the new array or null
    table = (tmp.length == 0) ? null : tmp;
  }
```

Figure 12: Cloned chunks copying an array minus one element at a given position: could be outlined into a final static method. The last statement is only matched with `struct` or `subAbstr`

java.util.Arrays.sort1(float[],int,int)

```
    /**
     * Sorts the specified sub−array of floats into ascending order.
     */
private static void
sort1 (float x[], int off, int len)
{
  // Insertion sort on smallest arrays
  if (len < 7)
    {
      for (int i = off; i < len + off; i++)
        for (int j = i; j > off && x[j − 1] > x[j]; j−−)
          swap (x, j, j − 1);
      return;
    }

  // Choose a partition element, v
  int m = off + (len >> 1); // Small arrays, middle element
  if (len > 7)
    {
      int l = off;
      int n = off + len − 1;
      if (len > 40)
        { // Big arrays, pseudomedian of 9
          int s = len / 8;
          l = med3 (x, l, l + s, l + 2 * s);
          m = med3 (x, m − s, m, m + s);
          n = med3 (x, n − 2 * s, n − s, n);
        }
      m = med3 (x, l, m, n); // Mid−size, med of 3
    }
  float v = x[m];

  // Establish Invariant: v* (<v)* (>v)* v*
  int a = off, b = a, c = off + len − 1, d = c;
  while (true)
    {
      while (b <= c && x[b] <= v)
        {
          if (x[b] == v)
            swap (x, a++, b);
          b++;
        }
      while (c >= b && x[c] >= v)
        {
          if (x[c] == v)
            swap (x, c, d−−);
          c−−;
        }
      if (b > c)
        break;
      swap (x, b++, c−−);
    }

  // Swap partition elements back to middle
  int s, n = off + len;
  s = Math.min (a − off, b − a);
  vecswap (x, off, b − s, s);
  s = Math.min (d − c, n − d − 1);
  vecswap (x, b, n − s, s);

  // Recursively sort non−partition−elements
  if ((s = b − a) > 1)
    sort1 (x, off, s);
  if ((s = d − c) > 1)
    sort1 (x, n − s, s);
}
```

java.util.Arrays.sort1(double[],int,int)

```
    /**
     * Sorts the specified sub−array of doubles into ascending order.
     */
private static void
sort1 (double x[], int off, int len)
{
  // Insertion sort on smallest arrays
  if (len < 7)
    {
      for (int i = off; i < len + off; i++)
        for (int j = i; j > off && x[j − 1] > x[j]; j−−)
          swap (x, j, j − 1);
      return;
    }

  // Choose a partition element, v
  int m = off + (len >> 1); // Small arrays, middle element
  if (len > 7)
    {
      int l = off;
      int n = off + len − 1;
      if (len > 40)
        { // Big arrays, pseudomedian of 9
          int s = len / 8;
          l = med3 (x, l, l + s, l + 2 * s);
          m = med3 (x, m − s, m, m + s);
          n = med3 (x, n − 2 * s, n − s, n);
        }
      m = med3 (x, l, m, n); // Mid−size, med of 3
    }
  double v = x[m];

  // Establish Invariant: v* (<v)* (>v)* v*
  int a = off, b = a, c = off + len − 1, d = c;
  while (true)
    {
      while (b <= c && x[b] <= v)
        {
          if (x[b] == v)
            swap (x, a++, b);
          b++;
        }
      while (c >= b && x[c] >= v)
        {
          if (x[c] == v)
            swap (x, c, d−−);
          c−−;
        }
      if (b > c)
        break;
      swap (x, b++, c−−);
    }

  // Swap partition elements back to middle
  int s, n = off + len;
  s = Math.min (a − off, b − a);
  vecswap (x, off, b − s, s);
  s = Math.min (d − c, n − d − 1);
  vecswap (x, b, n − s, s);

  // Recursively sort non−partition−elements
  if ((s = b − a) > 1)
    sort1 (x, off, s);
  if ((s = d − c) > 1)
    sort1 (x, n − s, s);
}
```

Figure 13: Two identical methods implementing sorting on different arrays of different primitive types. Cannot be refactored in Java without template support