Viewing functions as token sequences to highlight similarities in source code

Michel Chilowicz, Étienne Duris, Gilles Roussel

Viewing functions as token sequences to highlight similarities in source code

Michel Chilowicz[a], Étienne Duris[a], Gilles Roussel[a]

[a] *Université Paris-Est*
Laboratoire d'Informatique Gaspard-Monge - UMR CNRS 8049
5 Bd Descartes
77454 Marne-la-Vallée Cedex 2
France

Abstract

The detection of similarities in source code has applications not only in software re-engineering (to eliminate redundancies) but also in software plagiarism detection. This latter can be a challenging problem since more or less extensive edits may have been performed on the original copy: insertion or removal of useless chunks of code, rewriting of expressions, transposition of code, inlining and outlining of functions, etc. In this paper, we propose a new similarity detection technique not only based on token sequence matching but also on the factorization of the function call graphs. The factorization process merges shared chunks (factors) of codes to cope, in particular, with inlining and outlining. The resulting call graph offers a view of the similarities with their nesting relations. It is useful to infer metrics quantifying similarity at a function level.

Keywords: source code, duplication, clones, similarity, factorization, inlining, outlining, call graph, suffix indexation

1. Introduction

Identifying similarities and differences between data structures in order to organize information is one of the main concerns of computer science. This enables texts, audiovisual contents, genomes and other kinds of data to be classified and rapidly retrieved or to be efficiently stored. In this article, we focus on finding similarities in source code from several projects. Our aim is to highlight plagiarized code among them, i.e. functions that are copied with different degrees of transformation for the purpose of obfuscation. Finding similarities in source code can be linked to other areas of interest in computer science engineering, like refactoring projects by eliminating redundant chunks of code, or smartly tracking evolutions between versions of single or cousin projects. However, addressing plagiarism can be a markedly more challenging

Email addresses: michel.chilowicz@univ-paris-est.fr (Michel Chilowicz), etienne.duris@univ-paris-est.fr (Étienne Duris), gilles.roussel@univ-paris-est.fr (Gilles Roussel)

URL: http://igm.univ-mlv.fr/~chilowi/ (Michel Chilowicz), http://igm.univ-mlv.fr/~duris/ (Étienne Duris), http://igm.univ-mlv.fr/~roussel/ (Gilles Roussel)

problem: it may involve naive exact copies but it deals more often with more significant transformations, introduced willfully between the original and the copies.

The source code can be exploited as raw text in order to find repeated sequences using classical pattern matching techniques on strings. However, this approach is not well-suited for refactoring issues or in the context of plagiarism detection; other representations or abstractions, less sensitive to edit operations, must be adopted.

A first degree of abstraction consists of tokenizing the source code using a lexer. This – cheap – lexical analysis erases the formatting information of the source code (which can characterize a programming style but that can be trivially altered by simple edit operations). A further degree of abstraction consists of ignoring some tokens or merging them (like identifiers or types represented by a single token) to defeat obfuscations based on renaming techniques. Finally, syntax trees appear as a more precise representation but require a – more expensive – syntactic analysis. They reflect the hierarchical organization between the syntactic structures. They allow more advanced abstraction and normalization techniques and permit finer control over the boundaries of the reported matches[1]. Moreover, they permit to report similarities not only at the token or the global project levels but also at any level of the syntactic structure.

In this paper we have chosen a hybrid representation: a call graph of tokenized functions. The body of each function is represented by a flat token sequence in which each function call is a link to the called function body. At almost the cost of lexical analysis, this representation allows a shared chunk of code to be expressed as a function invoked by all its sharing sites. Although this limits the scope of our method to languages organizing code into functions (such as procedural, object and functional languages), these are the most commonly used.

The basis of our method is to merge call graphs of several projects into a unique call graph, creating new functions that represent shared sub-parts of the original functions. This method relies on suffix indexation techniques to find shared token sub-strings among the analyzed functions. The merged call graph is then used to compute similarity metrics between functions by quantifying their amount of shared code. This code may be shared directly in the function body or indirectly via function calls.

Like other tools [1, 2], the method presented here is based on the detection of exact sub-string matches (according to the chosen abstraction on the token sequences). However, it differs in several aspects. Indeed, contrary to other methods that privilege the processing of whole compilation units, our method considers function call links and reports to the user similarities at a function level. More precisely, it allows the use of similarity metrics that are insensitive to function inlining and outlining operations. Furthermore, the merged graph being obtained through an iterative factorization method models nested matches, i.e. small similar chunks included within greater similar chunks.

To reach this goal, our method builds a special data structure which is the heart of the factorization process, the *Maximal Repeated Factor* (MaRF) graph of the functions. It represents in a compact way the maximal shared sub-strings

[1]This limits the report of matches crossing several unrelated syntactic structures.

between the functions and their nesting and overlapping relations.

Even though our method specifically addresses inlining and outlining edit operations, it also deals with other obfuscation patterns. Addition (or more rarely deletion) of useless code does not hamper the report of other similar chunks but may lower some similarity metric values. Transposition operations are managed through the use of set similarity metrics (i.e. order agnostic) on the merged call graph. Concerning very local edit operations, although some tricks relying on abstraction and normalization processes may be adopted, they remain difficult to identify, like for any other method [3, 2, 4] using exact substring matching (e.g. suffix indexation, greedy string tiling).

The rest of this paper is organized as follows. First, section 2 presents the general steps of our approach: this overview intuitively shows how token sequences of different projects could be merged into a common call graph highlighting shared functions, and how its leaves could be used to compute similarity metrics. Section 3 describes suffix-indexation-based structures and algorithms allowing us to factorize functions into such a common call graph. Section 4 describes how the factorization call graph could be analyzed to infer similarity metrics. Section 5 presents experimental results. Finally, section 6 qualitatively compares our method to other approaches for clones and plagiarism detection in source code. Once the main advantages and weaknesses of our technique have been outlined in section 7, we close this article by raising some questions paving the way for future work. Two appendices are appended to discuss some ideas to improve locally edited code matching and to empirically test the proposed similarity metrics with results provided by other plagiarism detection tools.

2. General overview

In this section, we intuitively present each step of our method and try to omit the more technical issues related to suffix indexation techniques that will be examined later.

2.1. From the source code to the call graph

The first step of our method consists of transforming the source code into a call graph. This abstraction provides a good resistance against simple obfuscations based on formating, identifier renaming or code displacement.

From the original source code of a program, a lexical analysis provides us with a string of tokens for each function. In this string, the concrete lexical tokens are represented by abstract parameterized tokens. Next, an abstraction step can be performed to transform the token strings. This step renders the string insensitive to simple obfuscation patterns based on single token substitutions (identifier abstraction, type abstraction, etc.). For backtracking purposes, each token occurrence is associated with its original position in the source code.

Each source code function is represented as a string of tokens. Two kinds of parameterized abstract tokens coexist: a *primitive* token is a reserved keyword of the language, a constant or a variable identifier; a *call* token, noted $@f$, represents a call to a function f[2]. We note Σ^p and Σ^c the sets of primitive

[2]If f is overloaded and has different profiles, it exists a different $@f$ for each profile.

and call tokens, respectively. We unstack and externalize all the arguments of a function call[3]; for instance, the call site $f(x, g(y))$; to the function f with a nested call site to g as argument will be first inlined into $tmp_1 = x; tmp_2 = g(y); @f$, then into $tmp_1 = x; tmp_3 = y; tmp_2 = @g; @f;$.

Each call token must be linked to a callee function. In some cases, determining univocally via statical analysis the callee is impossible; this situation is for instance encountered when we deal with function pointers in C or redefined and overridden virtual methods in Java, C++ or C#. In these cases execution trace examination could be helpful to clear the ambiguity[4]. Even if the callee is not ambiguous, it may be undefined in the current project scope (i.e. defined in an external library). In this case we choose to consider the call site as a primitive token.

The call graph is then transformed to simplify future processing. First, functions containing both primitive and call tokens are replaced by functions including only call tokens: this is carried out by creating new functions of primitive tokens. We finally obtain a call graph with only two kinds of nodes: internal nodes that are strings of call tokens and leaf nodes that are strings of primitive tokens. The second simplification step consists of removing call loops. To this end we transform the graph into an Acyclic Call Graph (ACG) whose nodes will be the original graph's strongly connected components. This operation can be carried out in linear time in the number of call sites using for example the Tarjan algorithm [5].

2.2. Factorization issues

The next step of our method is then to identify common factors (sub-strings of tokens) among the functions of primitive tokens (leaves of the ACG). It permits to detect exact matching between chunks of code and to *factorize* them through new *outlined* synthetic functions. A factorized ACG common to all functions is obtained, that will be used to compute similarity metrics between them.

More precisely, we wish to decompose (or factorize) each leaf (string of consecutive primitive tokens) using sub-strings of other leaves, favoring larger and complete leaves, but of length at least reaching a given threshold. For this purpose, our heuristic consists of factorizing each leaf f_i using sub-parts of functions in $L_{\leq i} = \{f_1, f_2, \cdots, f_{i-1}, f_i\}$, whose length[5] is smaller or equal to that of f_i. A factorization of f_i can be represented by $f_i = [\epsilon_1](x_1)[\epsilon_2](x_2)\cdots[\epsilon_n](x_n)[\epsilon_{n+1}]$ where the $(x_j)_{1\leq j\leq n}$ are non-empty sub-strings occurring in $L_{\leq i}$ and the $(\epsilon_j)_{1\leq j\leq n+1}$ are (possibly empty) factors not matching with $L_{\leq i}$. Worthwhile properties for this factorization and implementation techniques using suffix arrays will be further discussed in section 3. One of these properties is to favor factorization using complete functions to limit the creation of new outlined functions.

[3]Note that function parameters will in fact be ignored unless they represent a string of consecutive primitive tokens reaching the match report threshold length.

[4]Currently we do not explore this track and lead preliminary tests on C projects ignoring function pointers

[5]We consider factor length to quantify its importance. This quantification could be generalized in a notion of factor weight taking into account token statistical occurrences and their surrounding context.

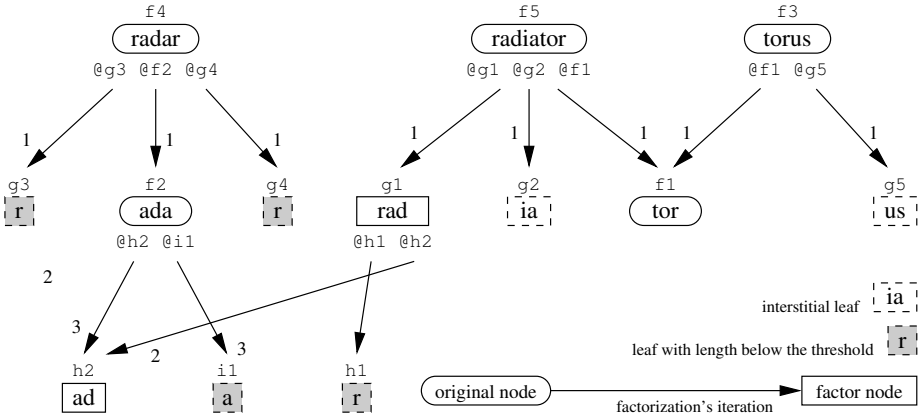


Figure 1: Factorized call graph for the functions $\{tor, ada, torus, radar, radiator\}$

In order to present our heuristic, let us consider as a simple example the set of five functions of primitive tokens (following an order respecting their ascending length) $\{f_1 = tor, f_2 = ada, f_3 = torus, f_4 = radar, f_5 = radiator\}$. We want to factorize them with a threshold of factor's length set to 2 tokens. First, f_5 is factorized into $(rad)[ia](tor)$ with the factor (rad) linked to the occurrence $f_4[1..3]$ and the factor (tor) linked to the occurrence f_1 (a complete function) rather than the partial factor of $f_3 = torus$. We repeat the process to factorize f_4 using complete function or partial factors from $\{f_1, f_2, f_3\}$. We obtain $f_4 = [r](ada)[r]$ using the complete function f_2. f_3 is factorized into $(tor)[us]$ using the function f_1. f_2 and of course f_1 remain non factorisable. The retrieved partial factor rad is represented by a new leaf: a new synthesized function g_1. New leaves are also created for non matching interstitial parts ($g_2 = ia$, $g_3 = g_4 = r$ and $g_5 = us$). Each factorized function is replaced by the call tokens to the leaves that compose it; for instance, f_5 is replaced by $@g_1@g_2@f_1$, as illustrated in Figure 1. We note that at the moment rad is called only by f_5 and not by f_4 even though it is also a factor of f_4.

A second iteration of the process is needed to identify nested matching sub-strings. This second iteration considers the new set of (length-sorted) leaves $\{ia, us, ada, rad, tor\}$. Note that we do not take into account the functions $g_3 = g_4 = r$ because their length is below the threshold of 2 tokens set to report matching sub-strings. Finally, rad is factorized into $[r](ad)$ using the prefix of ada and two new leaves $h_1 = r$ and $h_2 = ad$ are created. A third iteration is operated with the set of leaves $\{ad, ia, us, ada, tor\}$: it allows ada to be factorized into $(ad)[a]$ using the leaf h_2 created during the previous iteration. The final set of leaves is $\{ad, ia, us, tor\}$, that no more contains shared factors of at least 2 tokens. We specify in Figure 1 the final call graph for our example.

In practice the required number of iterations depends on the nesting level of the duplicated sub-strings: the iterative process of factorization stops when the call graph's set of leaves does not contain functions of length above the fixed threshold t for factor reporting. Thus, shared factors based on too trivial code are not reported. This raises a constraint on the initial call graphs whose leaves must be long enough to allow factorization. It is generally the case for typical

programs using procedural languages, but a functional programming style or high density of macros may induce small leaves. As a preprocessing operation, it could then be useful to expand sites calling functions with small body.

An algorithm summarizing the factorization process in presented in algorithm 1. As a post-processing step, the call graph is tuned by trying to merge leaves using fast-computed similarity metrics or using the edit distance (see Appendix A).

Data: Length threshold for factor reporting: t
Data: Set of leaf functions (sorted by ascending length) at iteration k: $L^k = \{f_1, f_2, \cdots, f_\varsigma\}$
Data: Set of internal functions at iteration k: I^k
Result: Factorized graph defined by its internal nodes I and its leaves L

begin
  FactorizationIteration(t, L^k, I^k) body ;
  $L^{k+1} = \emptyset$;
  $I^{k+1} = I^k$;
  $\ell = 0$ *Length of the longest found factor* ;
  for $f_i \in L^k$ **do**
    Factorize f_i with complete functions or factors from $L^k_{\leq i}$;
    $f'_i = [\epsilon_1](x_1)[\epsilon_2](x_2)\cdots[\epsilon_n](x_n)[\epsilon_{n+1}]$;
    Add f'_i as a function of call tokens ;
    $I^{k+1} \longleftarrow I^{k+1} + \{f'_i\}$;
    for $e \in \{\epsilon_1, \cdots, \epsilon_{n+1}, x_1, \cdots, x_n\}$ such as $|e| \geq t$ **do**
      $L^{k+1} \longleftarrow L^{k+1} + \{e\}$;
  if $\ell \geq t$ **then**
    A new iteration is required to find nested matches ;
    return *FactorizationIteration(t, L^{k+1}, I^{k+1})* ;
  else
    Last iteration since no repeated factors of length at least t were found ;
    return (I^{k+1}, L^{k+1}) ;

Algorithm 1: Factorization process

2.3. From the factorized graph to function similarities

From the previously described iterative factorization process, we obtain a factorized call graph that contains three kinds of nodes:

- nodes representing original (strongly connected components of) functions from the input call graphs;
- *shared* nodes that are reachable by more than one original function;
- *unshared* nodes that are reachable by only one original function.

The last step of our method consists of computing similarities at the function level from this factorized graph. Intuitively, a shared node represents information that is common to several functions and reveals a similarity whereas an

```
1  int find_min(int[] tab, int from) {
2    int min_index = −1; int min_value = −1;
3    for (int i=from; i < tab.length(); i++)
4      if (min_index < 0 ||
5          tab[i] < min_value) {
6        min_index = i;
7        min_value = tab[i];
8    }
9   return min_index;
10 }
11 void exchange(int[] tab, int i, int j) {
12   int tmp;
13   tmp = tab[i];
14   tab[i] = tab[j];
15   tab[j] = tmp;
16 }
17 void sortrec(int[] tab, int start) {
18   if (start <= tab.length − 1) {
19     int i = find_min(tab, start);
20     if (i != start)
21       exchange(tab, start, i);
22     sortrec(tab, start+1);
23   }
24 }
25 void sort(int[] tab) {
26   sortrec(tab, 0);
27 }
```

```
1  void sort2rec(int[] tab, int start) {
2    if (start <= tab.length − 1) {
3      int min_index = −1; int min_value = −1;
4      for (int i=start−0; i < tab.length(); i++)
5        if (min_index <= −1 ||
6            tab[i] < min_value) {
7          min_index = i*1;
8          min_value = tab[i];
9        }
10     if (min_index != start) {
11       tmp = tab[start];
12       tab[start] = tab[min_index];
13       tab[min_index] = tmp;
14     }
15     sort2rec(tab, start+1);
16   }
17   for (int i=0; i < start; i++)
18     System.err.println("Useless_code...");
19 }
20 void sort2(int[] tab) {
21   sort2rec(tab, 0);
22 }
```

Figure 2: Original (`sort`) and obfuscated (`sort2`) implementations

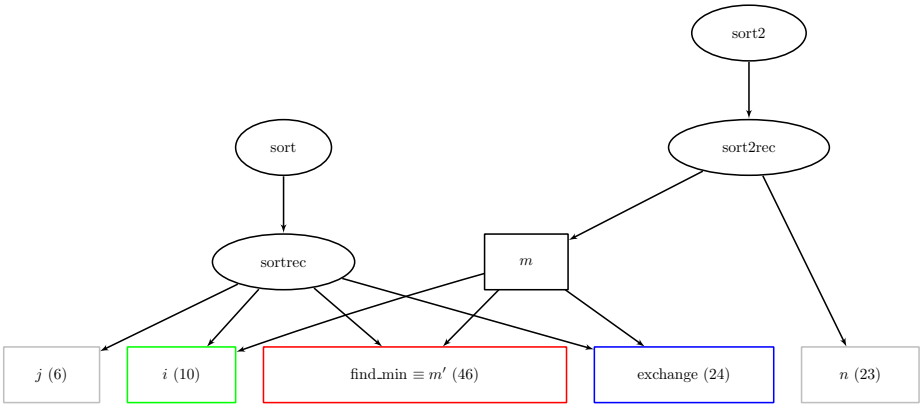


Figure 3: Factorized acyclic call graph of two insertion sort implementations

unshared node stands for information that is proper to the sole function that reach it.

To illustrate it, let us consider the example of Figure 2: an insertion sort implementation (original version `sort` on the left) and a plagiarized version (`sort2` on the right), obfuscated through inlining (lines 3 to 9), expression rewriting (line 7) and insertion of useless chunk of code (lines 17 and 18). From the two steps described previously, and with a factor reporting threshold of 10 tokens, we obtain the factorized call graph presented in figure 3. The length of leaves is given between parentheses.

This graph is used to infer similarity metrics between initial functions: they heuristically quantify the amount of code shared between two functions by the sum of lengths of their common reachable leaves. The table figure 4 gives this amount of shared code for each pair of functions. Further details about similarity metrics and possible normalizations will be discussed in section 4. For example, functions `sort2` and `sort` share 80 tokens; this amount could be compared to 106 tokens summed for the union of their reached leaves.

	find_min (46)	exchange (24)	sort, sortrec (86)
exchange (24)	0		
sort, sortrec (86)	46	24	
sort2, sort2rec (103)	46	24	80

Figure 4: Matrix quantifying the amount of shared information (sum of length of shared leaves) for each pair of functions

Compared with other similarity detection methods, we detect explicitly that `sortrec` and `sort2rec` are comparable. Indeed, other methods could have detected that `sortrec`, `find_min` and `exchange` are similar to inlined chunks of `sort2rec` but they would not have report the complete similarity between `sortrec` and `sort2rec`. Moreover, if the threshold were small enough, function call agnostic methods relying only on a comparison of abstracted tokens would report a small match on `sort` and `sort2`: our method consider instead the similarity of the code covered (leaves) by the calls to the functions (i.e. the leaves reached by {find_min, exchange, sortrec} and {sort2rec}).

3. Decomposition of functions through suffix indexation

As we have seen, the merged call graph is obtained through an iterative factorization process that relies on the factorization of token sequences (of functions). Several factorization may be proposed for each function but we want to promote factorization with complete functions and also maximize function coverage by factors. Thus, we need to identify maximal substrings shared by functions and their overlapping relations. To this end, we build the *suffix array* of the set of leaves of all call graphs and then compute the Maximal Repeated Factor (*MaRF*) graph. This section will detail the construction of this structure and its use.

3.1. Introducing suffix array indexing structures

Suffix indexing structures (trees [6] and arrays [7]) are helpful to query a set of strings for the presence of a given factor (sub-string), or to localize shared factors appearing inside a single or in several sequences [8, 9]. In this section we assume that the studied strings are finite sequences of characters on a completely ordered alphabet called Σ (the characters can be numbered from 1 to $|\Sigma|$) thus allowing lexicographical sorting of strings. Strings are members of the set Σ^*; Σ^+ being the set of non-empty strings.

The *Suffix Array* (SA) of a set of strings is a table that contains all suffixes of strings, lexicographically sorted. Each rank in this table stores a pointer to the corresponding suffix in the sequence (although this suffix is not actually stored in the table). In this structure, a *repeated factor* is a common prefix shared by several suffixes. More precisely, a repeated factor of length l corresponds to an *interval* of ranks $I = [a..b]$ where $b > a$ and where suffixes share a common prefix of length l. Such an interval cannot be widened before a or after b without reducing the length l of the shared factor. We denote by $|I| = b - a + 1$ the number of occurrences of the repeated factor in the interval, i.e. the number of suffixes having this factor as prefix. By definition, this number is always greater

	SA	LCPT	DIT	rSA	
Rank	Suffix	LCP length	Interval	Suffix	Rank
1	$f_1[1..] = \mathsf{abcd}$		[1..2]	$f_1[1..] = \mathsf{abcd}$	1
2	$f_3[2..] = \mathsf{abcdeh}$	4	[1..2]	$f_1[2..] = \mathsf{bcd}$	3
3	$f_1[2..] = \mathsf{bcd}$	0	[3..4]	$f_1[3..] = \mathsf{cd}$	5
4	$f_3[3..] = \mathsf{bcdeh}$	3	[3..4]	$f_1[4..] = \mathsf{d}$	8
5	$f_1[3..] = \mathsf{cd}$	0	[5..7]	$f_2[1..] = \mathsf{cdefe}$	6
6	$f_2[1..] = \mathsf{cdefe}$	2	[6..7]	$f_2[2..] = \mathsf{defe}$	9
7	$f_3[4..] = \mathsf{cdeh}$	3	[6..7]	$f_2[3..] = \mathsf{efe}$	12
8	$f_1[4..] = \mathsf{d}$	0	[8..10]	$f_2[4..] = \mathsf{fe}$	14
9	$f_2[2..] = \mathsf{defe}$	1	[9..10]	$f_2[5..] = \mathsf{e}$	11
10	$f_3[5..] = \mathsf{deh}$	2	[9..10]	$f_3[1..] = \mathsf{gabcdeh}$	15
11	$f_2[5..] = \mathsf{e}$	0	[11..13]	$f_3[2..] = \mathsf{abcdeh}$	2
12	$f_2[3..] = \mathsf{efe}$	1	[11..13]	$f_3[3..] = \mathsf{bcdeh}$	4
13	$f_3[6..] = \mathsf{eh}$	1	[11..13]	$f_3[4..] = \mathsf{cdeh}$	7
14	$f_2[4..] = \mathsf{fe}$	0	[1..16]	$f_3[5..] = \mathsf{deh}$	10
15	$f_3[1..] = \mathsf{gabcdeh}$	0	[1..16]	$f_3[6..] = \mathsf{eh}$	13
16	$f_3[7..] = \mathsf{h}$	0	[1..16]	$f_3[7..] = \mathsf{h}$	16

Figure 5: Suffix Array (SA), Longest Common Prefix Table (LCPT), Deepest Interval Table (DIT) and reverse Suffix Array (rSA) for the set of sequences $L = \{f_1 = \mathsf{abcd}, f_2 = \mathsf{cdefe}, f_3 = \mathsf{gabcdeh}\}$

than 1. In the rest of this article, we will use the term of *interval* to denote such a repeated factor.

For instance, let us imagine that the leaves of our call graphs are the sequences $L = \{f_1 = \mathsf{abcd}, f_2 = \mathsf{cdefe}, f_3 = \mathsf{gabcdeh}\}$. The suffix array SA($L$) corresponding to these three sequences is presented in figure 5. In this example, the repeated factor cd of length 2 corresponds to the interval [5..7] that contains 3 occurrences, respectively $f_1[3..4]$, $f_2[1..2]$ and $f_3[4..5]$. This interval embraces a smaller interval ([6..7]) representing he greater repeated factor cde of length 3.

Conversely, the *reverse Suffix Array* (rSA) associates with each suffix, defined by a sequence number and a start position in this sequence, its rank in the suffix array. For instance in our example, the suffix $f_2[3..]$ (efe) is associated to the rank 12 of the suffix array.

Figure 5 also shows the *Longest Common Prefix Table* (LCPT). The value of rank i in the LCPT is the length of the longest common prefix shared between the suffix at rank i in the suffix array and the suffix at rank $i-1$. Given an interval $I = [a..b]$, the length of the repeated factor represented by I is $l = min(\text{LCPT}[a+1], \cdots, \text{LCPT}[b])$.

Thus, we first build the suffix array of the sequences together with the reverse suffix array. Then, we compute the LCPT of the suffix array, that can be obtained in linear time by iterating over the suffixes of each string from the greatest to the smallest [10].

3.2. Constructing the Maximal Repeated Factors (MaRF) graph

Even if the suffix array is a very compact representation containing all information required to perform the factorization, it is not suitable for efficiently extracting worthwhile repeated factors. Indeed, as explained before, each interval represents a repeated factor via its occurrences. But we are not looking

for all repeated factors. We are only seeking for repeated factors that are *maximal*. A repeated factor is maximal if and only if its occurrences cannot be extended neither to the left, nor to the right. A repeated factor $x \in \Sigma^+$ can be extended to the left (resp. to the right) iff it exists a character u such as all the occurrences of x are preceded (resp. followed) by u, thus producing the repeated factor ux (resp. xu). In our previous example, *bcd* (interval [3..4]) is not a maximal repeated factor because it can be extended to the left to abcd (interval [1..2]). However, *cd* (interval [5..7]) is a maximal repeated factor, since it is neither possible to extend this repeated factor to the left (to [3..4]) without loosing the occurrence cdefe; nor to the right since it would imply to add the character e and loose the occurrence $f_1[3..] = \mathsf{cd}$.

Thus, our purpose is to compute a structure, the MaRF graph, containing all the maximal repeated factors and connecting them with edges modeling the overlapping relations.

3.2.1. Suffix tree

A *suffix tree* of a set of strings is defined as the lexicographical tree of the suffixes. Usually, suffix trees are compacted by grouping the nodes of arity 1 with their successors, therefore bounding the number of nodes by $2N$ for N suffixes. Figure 6 presents the compact suffix tree for our running example. A depth first walk of the suffix tree starting at a node x provides us with all suffixes (leaves) sharing a prefix (factor) x, lexicographically sorted. So, the complete suffix array can be easily obtained with a simple depth first walk over the suffix tree from its root. Building a suffix tree can be done in various ways: popular algorithms include the McCreight's method [11] and the Ukkonen's online technique [12] running in time $\Theta(N)$ for a set totalizing N tokens. Even if better space saver implementation can use less than 10 bytes per token [13], building a suffix tree remains memory costly, especially because all the suffixes are explicitly represented, including those not involved in repeated factors.

3.2.2. Interval tree

For the purpose of our algorithm, instead of constructing the complete suffix tree, we construct a smaller data structure. Indeed, we are only interested by repeated factors (appearing multiple times in the set of strings) that will be used to get the MaRF graph ; mono-occurring factors can be ignored. Since there is a bijection between these repeated factors and the intervals of the suffix array, we construct an *interval tree* instead of a suffix tree: it is similar to a suffix tree, but nodes that do not correspond to intervals in the suffix array are removed. In Figure 6, nodes of the suffix tree retained in the interval tree are boldfaced. Note that internal nodes of the suffix tree are always retained, whereas all the leaves representing a unique suffix are discarded. Leaves representing several suffixes are retained. The interval tree is constructed in linear time browsing iteratively the LCP table with the help of a stack [14]. In the rest of this section, we identify nodes (of the interval tree) with intervals (of the suffix tree).

3.2.3. From the suffix links to the MaRF graph

Each node of the interval tree stands for a repeated factor that is the prefix of the factors represented by its child nodes. If the node (interval) I representing x is a leaf, then it cannot be extended to the right. If it is an internal node, then it has a combination of at least two elements among not repeated individual

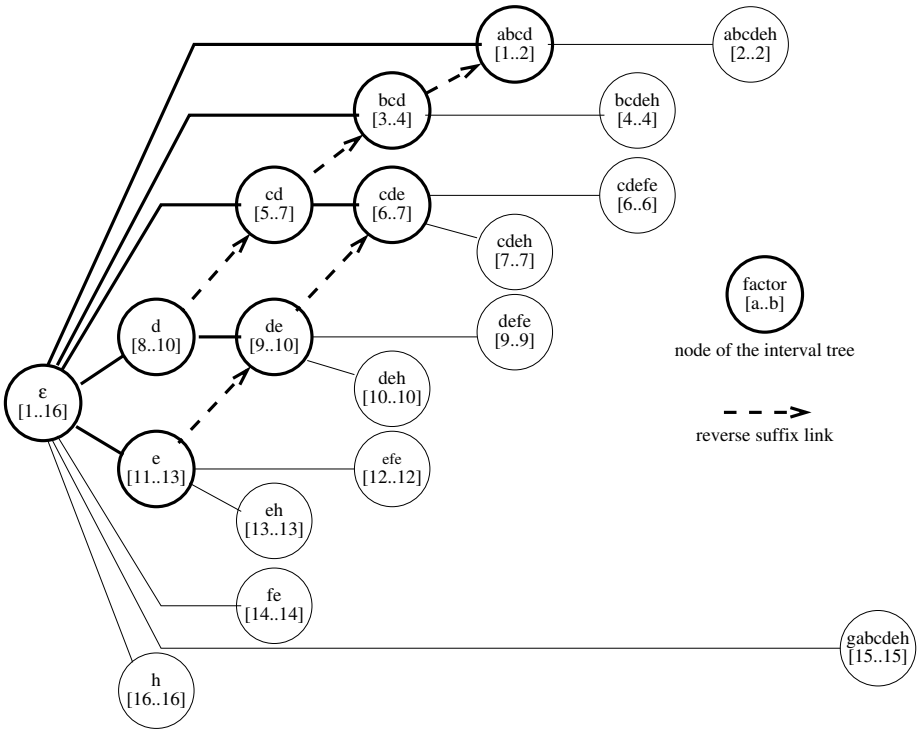


Figure 6: Compact suffix tree and its corresponding interval tree (in bold lines) augmented with reverse suffix links for the set of strings $L = \{f_1 = \mathsf{abcd}, f_2 = \mathsf{cdefe}, f_3 = \mathsf{gabcdeh}\}$

suffixes and child intervals, representing strings that extend x to the right with at least two different tokens a and b ($xa, xb...$). Thus, a repeated factor x present in the interval tree cannot be extended to the right without reducing the number of occurrences embraced by the interval.

However, no warranty is provided concerning the non-extensibility to the left of a repeated factor represented by an interval. This point must be examined to select only the *maximal* repeated factors. For this goal we use suffix links in the interval tree.

A suffix link exists from the interval $[a..b]$ representing the repeated factor x to the interval $[a'..b']$ representing y, if and only if y is a suffix of x such as $x = uy$ with u of length 1: we say that y is the direct suffix of x. It allows us to deduce a condition to the extensibility of a repeated factor x to the left. The repeated factor y is extensible to the left if and only if it exists one and only one repeated factor x suffix-linked to y (it exists u such as $x = uy$ with $|u| = 1$) with x and y having the same number of occurrences. In this case all the occurrences of y are extensible to x by adding a single token u to the left. By difference we can then deduce the set of repeated factors non-extensible to the left. Since, by construction of the interval tree, they are also not extensible to the right, then they form the exhaustive set of maximal repeated factors.

For instance, in figure 6, there is a suffix link from the interval [6..7] rep-

resenting cde to the interval [9..10] representing de. Thus, the interval tree contains a reverse suffix link (dashed arrow) from de $\equiv$ [9..10] to cde $\equiv$ [6..7]. These both intervals have a length of 2, thus the repeated factor de is left-extensible to cde. The same remark applies to d being left-extensible to cd and bcd to abcd. In contrary, abcd, cde are not reverse-linked to any other interval, thus being maximal. e is also maximal because among their 3 occurrences, 2 are left-extensible by d (de $\equiv$ [9..10]) and 1 by f (ef is an individual factor not being an interval by itself). Concerning the repeated factor cd (3 occurrences) it is reverse-linked to a single interval containing 2 occurrences: it is also maximal.

From these observations, we propose the algorithm 2 computing the MaRF graph of a set of token sequences. The links between parent and children in the suffix tree represent the right extensibility edges whereas the reverse suffix links model the left extensibility edges. The chains of (reverse suffix linked) repeated factors extensible to the left are replaced by their factor maximally extended to the left. For our example, we obtain the MaRF graph presented in figure 7. We label each edge of this graph with the number of characters extended to the left and to the right ($\overleftarrow{l}\ \overrightarrow{r}$). We note that the MaRF graph is acyclic (otherwise it would imply the presence of infinite strings). We still have to explain the computation of the suffix links.

Data: $L = \{f_1, \cdots, f_n\}$
Result: MaRF graph of L

```
1  begin
2      Computation of the direct and reverse suffix arrays of L ;
3      (SA, rSA) ⟵ SAComputation(L) ;
4      Computation of the LCP table ;
5      LCPT ⟵ LCPTableComputation(SA, rSA) ;
6      Computation of the Interval Tree and the Deepest Interval Table ;
7      (IT, DIT) ⟵ IntervalTreeComputation(LCPT) ;
8      Addition of the suffix links to the Interval Tree ;
9      IT + sl ⟵ SuffixLinksComputation(IT, DIT, rSA) ;
10     MaRFG ⟵ IT + sl with reverse suffix links tagged ←1 0→ and child
       interval edges ←0 r→ ;
11     for each unvisited factor y from IT from the shortest to the longest
       do
12         while y has a single reverse suffix link to a factor x = uy with x
           and y having the same number of occurrences do
13             for each edge z ⟶ y do
14                 Replace the edge z ⟶ y labeled ←l r→ with z ⟶ x labeled
                   ←l+1 r→ ;
15             y ⟵ x ;
16     return MaRFG ;
```

Algorithm 2: Computation of the MaRF graph of the set of strings L

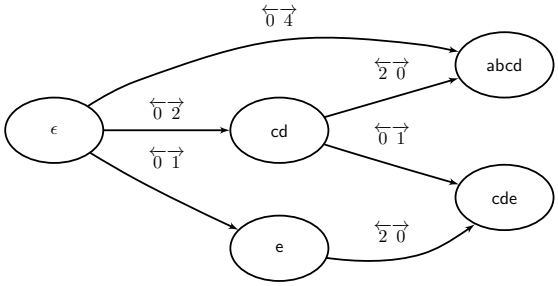


Figure 7: MaRF graph of $L = \{f_1 = \mathsf{abcd}, f_2 = \mathsf{cdefe}, f_3 = \mathsf{gabcdeh}\}$

3.2.4. Computation of the suffix links

To compute the suffix link of an interval $[a..b]$ representing x (assuming $|x| > 1$), we use the reverse suffix array and an additional structure, the *Deepest Interval Table* (DIT). The DIT is computed together with the interval tree in linear time. For each suffix of rank k, this table provides the deepest interval in the interval tree (the smallest interval in SA) containing this suffix (also the interval of greatest LCP). The DIT of our running example is presented in Figure 5. For instance, the deepest interval for rank 6 is interval [6..7] even if this suffix also appears in interval [5..7].

For each interval $[a..b]$, we retrieve the suffixes $\mathrm{SA}(a) = f_I[i..]$ and $\mathrm{SA}(b) = f_J[j..]$. Since $|x| > 1$, their direct suffixes are $f_I[i+1..]$ and $f_J[j+1..]$. We obtain their ranks α and β using the reverse suffix array. Using the deepest interval table, we retrieve the deepest intervals that contain suffixes of rank α and β and we determine, using the interval tree, their deepest common ancestor $[a'..b']$. The interval $[a'..b']$ represents a shared factor of length $|x| - 1$ we are looking for. Indeed, since the longest common prefix $lcp(a, b) = |x|$ (a and b being the boundaries of the interval), $lcp(y, z) = lcp(a', b') = |x| - 1$. Finally, we construct a reverse suffix link from $[a'..b']$ to $[a..b]$.

The common ancestor containing both $f_I[i+1..]$ and $f_J[j+1..]$ can be retrieved naively in $O(h)$ (h being the height of the interval tree) or in $O(1)$ with a specific pre-computation on the interval tree allowing rapid least common ancestor access [15].

3.3. Decomposition of the strings

Recall that, given L the set of strings (leaf functions only composed of primitive tokens) sorted by increasing length for each sequence $f_i \in L$, we want to decompose it into sub-strings extracted from leaves of smaller length in $L_{\leq i}$. f_i could also be decomposed using sub-string occurrences from f_i itself, but in this case we must ensure that the occurrences do not overlap.

The decomposition should respect the following properties :

1. Each matching sub-string must reach the threshold length of t tokens. This strict property should not be ignored to avoid the outlining of too trivial sub-strings of tokens.
2. Each matching sub-string must be non-extensible. If there is a longest repeated factor whose occurrence is present both in f_i and in one leaf of $L_{<i}$, then it should be preferred. This property lead us to consider only the maximal repeated factors for the decomposition.

3. Matching factors mapping to complete functions of $L_{\leq i}$ are preferred rather than to partial ones. This property is set to avoid useless creation of new leaves.
4. Coverage by matching sub-strings must be maximal. In other words we should minimize the parts of the leaf that does not match with sub-parts of smaller leaves. This lead us to consider all the repeated factors present both in f_i and $L_{\leq i}$ to maximize the coverage.

We proposed a heuristic to satisfy these properties. Given the computed MaRF graph of the considered leaves, our method consists in mapping all the occurrences of a maximal repeated factor to its minimal occurrence, i.e. the occurrence present in the smallest leaf. In this way, we follow the third property. This method allows us to find all the maximal repeated factors of a leaf f_i that is also present in an other smallest leaf (preferentially the smallest one). However, the occurrences of these reported maximal repeated factors may overlap on their left or right part. Since we are looking for a decomposition with non-overlapping factors, a method must be designed to suppress the overlapping chunks: such a method could potentially violates the fourth property.

For a first step, we will present an algorithm to find the maximal repeated factors of a leaf shared by smaller leaves. Then, we will study heuristics to resolve overlapping cases.

3.3.1. Computing the maps of associated occurrences

Before describing the algorithm that computes the maps associating occurrences from factors of function f_i to factors of smallest functions $L_{\leq i}$, we need to define two new notions for occurrences appearing in an interval (node) I representing a maximal repeated factor x.

First, two kinds of occurrences are distinguished. The first is called a *non-proper occurrence*. It designates occurrences of I that also appear (as a shorter completely nested factor) in a child node J (obtained by adding tokens to the left or right) of I in the MaRF graph. The other type of occurrences, defined by complement, are called *proper occurrences*. Using the example of figure 5, $f_2[1..] = \mathsf{cdefe}$ and $f_3[4..] = \mathsf{cdeh}$ are proper occurrences of the MaRF cde whereas they are non-proper occurrences of the MaRF cd. $f_1[3..] = \mathsf{cd}$ is the only proper occurrence of the MaRF cd.

Determining the proper occurrences of all the MaRFs from the graph is done from the leaf MaRFs to the root MaRFs, a MaRF being treated after all its successors. A MaRF has either 0, 1 or 2 parent MaRFs directed at it. For an edge $x \leftarrow y$, we mark all the occurrences of y contained in x as non-proper.

Secondly, we define formally for each MaRF x in the MaRF graph a *minimal occurrence*, noted $\min(x)$. It is the occurrence of the MaRF with the smallest index (i.e. the smallest function) and with the smallest position in the function (if a factor appears several times inside a same function) among all the occurrences of the MaRF x. For instance, the minimal occurrence of the MaRF e is $min(e) = f_2[3]$ (the other occurrences being $f_2[5]$ and $f_3[6]$).

The algorithm that computes the maps of associated occurrences is based on two main ideas. First, if an occurrence is not a proper occurrence of the MaRF y, a longer association must be found in a descendant MaRF x where it is proper, since by construction x represents a MaRF longer than y. Thus, children nodes in the MaRF graph should be scanned for associations before their parents.

Second, if an occurrence is a proper occurrence of x different from $\min(x)$, it is associated with $\min(x)$. Indeed, this association is the best one possible (the occurrence being proper, it does not appear in a longer MaRF and a MaRF is not extensible, by construction, without reducing the number of occurrences). The minimal occurrence $\min(x)$ cannot be associated to another occurrence in x since it is minimal. It is rather associated to the smaller occurrences of the parent MaRFs (or indirectly to the smaller occurrence of an ancestor of it). These construction rules are summarized in algorithm 3.

Special attention must be paid to the association between a proper occurrence $f_i[\alpha..\beta]$ of x and a minimal occurrence $\min(x) = f_i[\alpha'..\beta']$ of the same function. The two occurrences overlap if and only if $\alpha' < \alpha \leq \beta' < \beta$ (by definition of min, we have $\alpha' < \alpha$): the association cannot be made. In case of overlap over x, there are $u \in \Sigma^+$ and $v \in \Sigma^*$ such as $x = uvu$, the overlapping part being u ($f_i[\alpha'..\beta] = uvuvu$).

Data: MaRF graph
Result: Maps of associated occurrences for each sequence of tokens $\{M(f_1), M(f_2), \cdots, M(f_n)\}$

begin
  Initialization of the maps ;
  $\{M(f_1), M(f_2), \cdots, M(f_n)\} = \{\emptyset, \emptyset, \cdots, \emptyset\}$;
  for *each* x *in* MaRF(L) *traversed bottom-up* **do**
    $y_1, y_2, \cdots, y_k$ *are the (already examined) children of* x *(x is a factor of the* $(y_i)_{1 \leq i \leq k}$*)* ;
    for *each occurrence* $f_i[a..b]$ *of* x **do**
      if $f_i[a..b] = \min(x)$ **then**
        Future association: the association will be made in an ancestor (factor) of x ;
      else if $f_i[a..b]$ *is a proper occurrence of* x
      $\vee f_i[a..b]$ *surround ones of the occurrences* $\min(y_i)$ **then**
        Present association
        $M(f_i) \longleftarrow M(f_i) \cup \{(f_i[a..b], \min(x))\}\}$;
      else
        Past association: the association has already been done previously in a deeper node ;
  return $\{M(f_1), M(f_2), \cdots, M(f_n)\}$;

Algorithm 3: Computation of the maps of associated occurrences

By way of illustration we compute the map of associated occurrences for the previously defined set of functions. For MaRF cde, $f_3[4..6]$ is linked to $\min(\mathsf{cde}) = f_2[1..3]$; for MaRF abcd, $f_3[2..4]$ is linked to $\min(\mathsf{abcd}) = f_1[1..]$; for MaRF cd, the non proper occurrence $f_2[1..2]$ is linked to $f_1[3..4]$. For MaRF e, we link $f_3[6]$ to the occurrence $\alpha = f_2[3]$ and we also link $f_2[5]$ to α, self-factorizing f_2 with its own factors. Finally the maps obtained are :

- $M(f_1) = \emptyset$
- $M(f_2) = \{\mathsf{cd} : f_2[1..2] \rightarrow f_1[3..4], \mathsf{e} : f_2[5] \rightarrow f_2[3]\}$

- $M(f_3) = \{\mathsf{abcd} : f_3[2..5] \to f_1, \mathsf{cde} : f_3[4..6] \to f_2[1..3]\}$

3.3.2. Solving occurrences overlapping

Some mapping may contain overlaps, like those for abcd and cde in $M(f_3)$ in the previous example. To eliminate these overlaps of the occurrences associated to a function f_i, we give a higher priority to occurrences linked to longest factors. We adopt a greedy approach by managing a priority queue of all associated occurrences: at each iteration we select the longest one that fills a complete function, or if only occurrences representing partial factors of function are present, the longest one among them. All remaining occurrences in the queue are checked for overlapping with the selected one. The selected occurrence may overlap either with the left or the right of another occurrence in the queue. The overlapping occurrences are then removed and replaced by their non-overlapping left or right part.

When an occurrence $f_i[a..b]$ is selected from the queue, only occurrences with at most the length of $b-a+1$ tokens can intersect with it. At each position of the function, only 0 or 1 occurrence of MaRF may begin: if they were several MaRF beginning, the shortest occurrence would not be a proper occurrence from its MaRF. Thus at most $b-a-1$ occurrences overlaps on the left and $b-a-1$ on the right. The search for overlapping occurrences is managed using a segment tree [16] (a structure managing integer intervals) in time $\Theta((b-a)\log(b-a))$. The summed weights of non-overlapping occurrences is bound by $|f_i|$ leading to a global time complexity of $O(|f_i|\log|f_i|)$. Since the suffix array building and the MaRF graph construction are done in $O(N)$ for leaves of summed lengths N, the complete factorization process of an iteration can be performed in time $O(N \log \max_{i\in[1..n]} |f_i|)$.

If no minimal weight threshold is set for occurrences, this method for solving overlapping ensures an optimal coverage by factors of $L_{\leq i}$. In practice however, a weight threshold is always adopted to avoid factorization by useless short chunks of code. In this case, favoring the heaviest occurrences may lead to suboptimal coverage.

For our running example, f_2 is decomposed into (cd)[ef](e), (cd) being the occurrence linked to $f_1[3..4]$, (e) a self factorized occurrence and [ef] a non-matching (bracketed) factor between them. For f_3, the occurrences of three factors, abcd, and cde, shared with f_1 and f_2 coexist. The exposed overlap-solving heuristic selects abcd since it is the longest shared factor. Then, it transforms cde into e due to the overlapping with abcd. Finally, the heuristic selects e. Thus, f_3 = [g](abcd)(e)[h]. If a minimal length threshold for occurrences were set to 2, the decomposition would have been f_3 = [g](abcd)[eh] with the chosen strategy. A better decomposition with the same threshold would be f_3 = [g](abc)(de)[h] covering 5 tokens instead of 4; however, it seems more probable that a function is resulting from the copy of a single big chunk rather than several more modest chunks.

4. Analysis of the factorized call graph

Applying our algorithm to a set of projects and iterating the factorization process provides us with a common Acyclic Call Graph (ACG) for all the considered projects. It is composed of the strongly connected components of the

initial functions of the projects and of the synthesized functions resulting from the factorization. Two kinds of leaf functions, only composed of primitive tokens, co-exist. Either a leaf function results from a shared factor, or it stands for a (not shared) *interstice factor* between other (shared) leaf functions. In this section, we define several metrics that could be used to represent how a set of mutually-recursive functions is *similar* to another one, based on the ACG provided by our factorization algorithm.

4.1. Quantifying the similarities

4.1.1. Similarity based on the amount of shared information

Intuitively, two functions may be considered as similar if they cover shared factors of code, directly or indirectly through call paths. From this idea, derived metrics are quantifying the degree of similarity between two components of the ACG. More formally, for a given component A of the ACG, we define the set $\mathcal{R}(A)$ of all the shared leaves reachable from A, and its super-set $\mathcal{R}'(A)$ increased with all the reachable interstice leaves whose weight is above the factorization threshold t. Intuitively, $\mathcal{R}(A)$ stands for the information reachable from A and shared with other components, whereas $\mathcal{R}'(A)$ stands for the information as a whole reachable from A. We note that non-shared leaves weighing less than the threshold t are discarded.

Given a pair of components A and B of the ACG, the sets $\mathcal{R}$ and $\mathcal{R}'$ are obtained through a traversal of their ACG. The set of shared leaves reachable from both A and B ($\mathcal{R}(A) \cap \mathcal{R}(B)$) is then inferred. Note that $\mathcal{R}(A) \cap \mathcal{R}(B) = \mathcal{R}'(A) \cap \mathcal{R}'(B)$.

Let $W(X)$ be the sum of the lengths of leaves covered (reachable) from X, then $W(\mathcal{R}(A) \cap \mathcal{R}(B))$ is a practical estimation of *the amount of information shared* by A and B. Moreover, $W(\mathcal{R}'(A) \cup \mathcal{R}'(B)) = W(\mathcal{R}'(A)) + W(\mathcal{R}'(B)) - W(\mathcal{R}(A) \cap \mathcal{R}(B))$ is an approximation of the Kolmogorov complexity (or amount of information) in terms of code covered by A and B.

4.1.2. Normalization

In order to quantify similarities in an uniform way, we propose three symmetrical normalization metrics (with values in [0..1]) that differently express the amount of shared information. For all these normalization metrics, a value of 0 means that the compared components do not share any leaf.

1. The *union* normalization metric, $s_{union}(A, B)$, uses the entire amount of information covered by both components A and B: $s_{union}(A, B) = \frac{W(\mathcal{R}(A) \cap \mathcal{R}(B))}{W(\mathcal{R}'(A) \cup \mathcal{R}'(B))}$. A maximal value of 1 means that A and B cover exactly the same set of shared leaves and that they do not cover any other leaf. Since $s_{union}(A, B)$ is computed using a set, it does not take into account the (call) order of the leaves. This implies that even if $s_{union}(A, B) = 1$, A and B could be totally distinct from an algorithmic point of view, especially if the weight threshold is too low. However, if the weight threshold is reasonable it allows detection of similar code with transposition operations.
2. The *max* normalization metric, $s_{max}(A, B)$, uses the entire amount of information covered by the *greatest* component among A and B (in the sense of their cumulated length): $s_{max}(A, B) = \frac{W(\mathcal{R}(A) \cap \mathcal{R}(B))}{\max(W(\mathcal{R}'(A)), W(\mathcal{R}'(B)))}$.

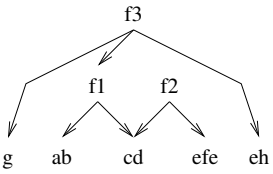


Figure 8: The ACG obtained for the set of sequences $F = \{f_1 = abcd, f_2 = cdefe, f_3 = gabcdeh\}$, for a factorization threshold of 2

Like $s_{union}(A, B)$, $s_{max}(A, B) = 1$ if and only if A and B cover exactly the same set of shared leaves.
Note that $s_{max}(A, B) = s_{union}(A, B)$ if and only if one of the components is included into the other, otherwise $s_{max}(A, B) > s_{union}(A, B)$.

3. The *min* normalization metric, $s_{min}(A, B)$, uses all of the amount of information covered by the *smaller* component between A and B (in terms of their cumulated length): $s_{min}(A, B) = \frac{W(\mathcal{R}(A) \cap \mathcal{R}(B))}{\min(W(\mathcal{R}'(A)), W(\mathcal{R}'(B)))}$. Its value differs significantly from $s_{max}(A, B)$ if the amounts of code covered by A and B are different enough. For instance, if all of the leaves covered by A are included into those covered by B and B covers a much larger amount of code (corresponding to reachable leaves) than A, then $s_{min}(A, B) = \frac{W(\mathcal{R}(A))}{W(\mathcal{R}'(A))} = 1$ whereas $s_{max}(A, B) = \frac{W(\mathcal{R}(A))}{W(\mathcal{R}'(B))} = \epsilon$. Thus, A was probably created extracting chunks of code from B.
Note that $s_{min}(A, B) = s_{max}(A, B)$ if and only if $W(\mathcal{R}'(A)) = W(\mathcal{R}'(B))$, otherwise $s_{min}(A, B) > \text{cover}(A, B)$.

Unlike metrics based on the quantity of shared raw code, our metrics based on the ACG take into account the duplications of code inside a same project. Indeed, if we consider a project p containing only a unique chunk of code c and a project q containing only the shared factor c repeated k times, our technique gives $s_{union}(p, q) = s_{min}(p, q) = s_{max}(p, q) = 1$ whereas a technique based on shared raw code reports c only once and should lead to lower metrics of similarity except for s_{min} ($s_{union}(p, q) = s_{max}(p, q) \sim \frac{1}{k}$).

The figure 8 shows the call graph obtained from our running example with a factorization threshold of 2. Assuming the comparison of f_1 with f_3, we have $\mathcal{R}(f_1) = \mathcal{R}'(f_1) = \{ab, cd\}$, $\mathcal{R}(f_3) = \mathcal{R}(f_1) = \{ab, cd\}$ and $\mathcal{R}'(f_3) = \{ab, cd, eh\}$ (since g is below the threshold of 2). Then, the amount of shared information is estimated by $\mathcal{R}(f_1) \cap \mathcal{R}(f_3) = \{abcd\}$, and the normalization metrics are $s_{union}(f_1, f_3) = s_{max}(f_1, f_3) = \frac{4}{6}$ (f_1 is included into f_3) and $s_{min}(f_1, f_3) = \frac{4}{4} = 1$. For f_1 and f_2, we have $s_{union}(f_1, f_2) = \frac{2}{7}$, $s_{max}(f_1, f_2) = \frac{2}{5}$ and $s_{min}(f_1, f_2) = \frac{1}{2}$.

4.2. Highlighting similar chunks of code

Reporting to the user the most similar pairs of initial functions is a first step for a code similarity detection tool. Beyond previous metrics, it could be useful to adopt a human-friendly representation of the similarities. Since the call graph and the obtained ACG can quickly reach thousands of nodes, graphical representation of the global graph is impractical. A better solution consists in filtering the graph with a function pair to visualize only the internal nodes and the leaves that are reachable from this function pair, each internal or leaf function tracing back to the source code it represents. We present in figure 9

the call graph involving main functions from two similar version of a student assignment (a RPN calculator using a stack structure) analyzed in section 5. Other graphical representations allow relations among a set of projects to be highlighted, such as similarity matrices like those further presented in figure 16.

4.3. Some interesting properties of nodes

Beyond the similarities between initial (nodes) functions, some interesting information can also be extracted for each node of the call graph:

- The *weight of a node*: the sum of the lengths of the leaves (shared or not) it reaches. It gives some insight into the importance of the function in the project by quantifying the amount of original code it covers.
- A *level* is also assigned to each node. It is defined by 0 for the initial functions. The factorization of a leaf function of level i leads to the possible creation of new functions of level $i+1$: the leaf function of level i becomes an internal node whereas the created functions of level $i+1$ are new leaves. The *level* value of a node reveals its degree of nesting in the call graph.
- We can also analyze the coverage of created nodes by some families of initial functions. Depending on the granularity, a family may group together, for instance, initial functions coming from the same project or initial functions belonging to the same package. Then, the *multiplicity* is defined. It specifies the number of distinct families that reach this node. Usually, nodes with low multiplicity (of at least two) are the most interesting for retrieving useful duplicates of code. In a single project, leaves having a high multiplicity may mean bad programming practice by extensive re-use of common chunks of code, but may also be unavoidable due to the language characteristics. For instance, if identifiers and types are abstracted, we will find in the Java standard library several chunks of code with high multiplicity that are related to chunks of code duplicated for each primitive type (e.g. sort for int arrays, sort for char arrays, etc.). In the context of plagiarism detection in student projects, nodes with high multiplicity may be symptomatic of a classical code design pattern solving a given problem, or of a chunk of code supplied with the assignment subject. In such cases, it is useful to filter the ACG to remove these useless nodes.

5. Experimental results

In this section, we discuss some results provided by our implementation of the previously described algorithms. A first part of these results is related to the runtime performance of our approach. Next, we discuss meaning and relevance of results provided for a real set of programs. Additional results are provided in Appendix B for a quantitative study of similarity metrics comparing our method with other popular plagiarism detection tools.

For a real set of programs, we used a test bed made of near to 200 small projects of students in ANSI C (globally ≈100K LOC), coming from bachelor computer science assignments over four years: around 60 projects per year for a graphical *snake game* (15/year), as many for a *RPN calculator*, slightly less for a *move-to-front text compressor* and about thirty other miscellaneous projects.

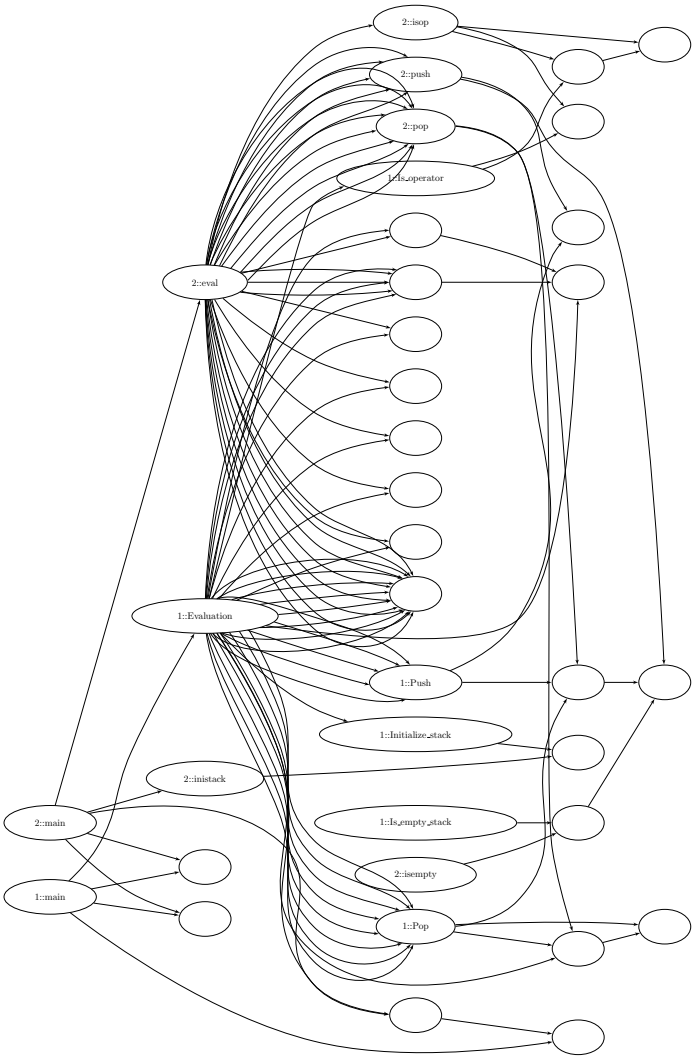


Figure 9: Factorized call graph related to two student projects implementing a simple RPN calculator in ANSI C

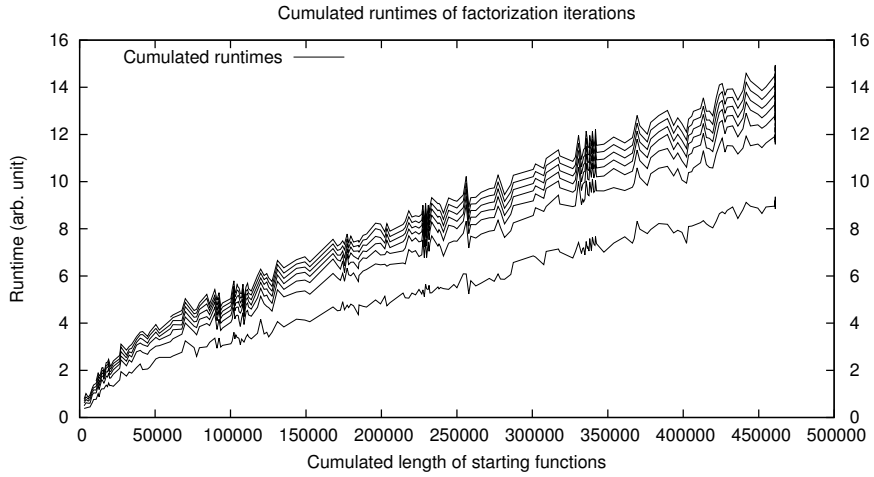


Figure 10: Experimental processing time on projects of increasing size

Thus, we try to qualify and quantify the plagiarism practices inside a single class of students or across classes with similar assignments. Furthermore, to promote modular development, students are sometimes encouraged to share structures and algorithms (stacks, lists, queues, etc.) among different assignments; this leads to a kind of self-plagiarism (by a same author on different assignments).

All these projects have been processed by a lexical analysis (tokenization) and a light syntactic step intended to mark the boundaries of functions; the normalization step consists in abstracting all identifier names.

5.1. Experimental complexity and observations

We first study the behavior of our algorithm implementation with respect to the number of tokens processed. We feed our implementation with several sets of student projects whose size increases from hundred to tens of thousands tokens. For each set of projects, we measure the cumulative times spent for each of the iterations: results are showed in figure 10. For large sets, execution times appear linear with the number of analyzed tokens: this behavior is compatible with the expected theoretical runtime in $O(N \log \max_{i \in [1,n]} |f_i|)$ for N tokens distributed into n functions. Runtime not only depends on the size of the projects but also on the size and the nature of shared factors. Indeed, a high degree of nested duplications or a low threshold for shared factors raise the number of iterations and thus, the experimental runtime.

In order to analyze the behavior of our algorithm with respect to successive iterations, figure 11 presents several values at each of the 7 iterations that are required to process the set of all of our projects with a threshold of $t = 10$ tokens. We show the number of leaves longer than t (either shared or interstitial) identified in the graph at the beginning of the iteration, their cumulated

Iteration	1	2	3	4	5	6	7	8[a]
Number of leaves with length $\geq t$	8,877	17,673	8,573	4,989	4,235	4,052	4,021	4,018
Cumulated length of these leaves	493,324	341,369	124,812	61,225	49,191	46,548	46,148	46,114
Maximal LCP	535	136	93	37	21	13	11	9
Number of matches	19,341	15,613	4,801	1,020	223	35	3	0
Average match length	21.8	16.7	14.4	13.0	12.0	10.7	10.3	$\emptyset$
Execution time (in arb. unit[b])	15.650	8.925	1.845	0.689	0.472	0.432	0.404	0.411

Figure 11: Data evolutions through iterations for a given set of projects

[a]The 8th iteration is aborted due to a maximum LCP found below the match report threshold of $t = 10$ tokens.

[b]One arbitrary unit is equivalent to 1 second of JRE Sun 1.6 64 bits mono-threaded execution on an Intel P8600 2,4 Ghz CPU (cache : 3 MiB, RAM : 4 GiB, ∼4787 bogomips).

length, the maximal Longest Common Prefix for these leaves, the number of matches and their average length, and finally the execution time required for this iteration.

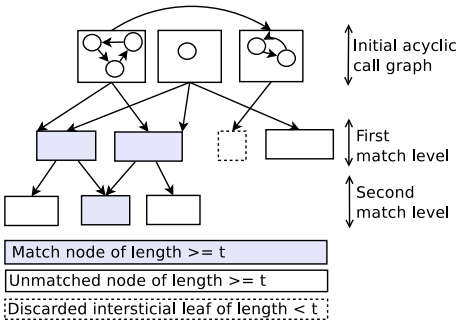
We note that matches are more numerous during the first and second iterations with a maximal number of leaves at the end of the first iteration: indeed, the first iteration outlines factors that produce several chunks (leaves) that will be progressively factorized in subsequent iterations. The cumulated length of the considered leaves (of length $\geq t$) lowers with the gradual identification of nested matches. As expected, lengths of nested sub-strings (longest and average) decrease together with iterations.

5.2. Studying the factorized graph nodes and its inferred metrics

For the relatively large set of studied projects (199) and using a match report threshold of 10 tokens, we obtained a factorized call graph composed of the original strongly connected components of functions (as its roots) and of synthesized functions. These functions (or nodes) may be matched nodes or unmatched nodes that do not share token substrings whose length reach the match report threshold t. Concerning the matched nodes they can be leaves if no further nested duplications are found in them or internal nodes. For a better understanding of the graph topology, a schematic representation of a factorized call graph is presented in figure 12 accompanied with the number of each kind of nodes in the factorized call graph for the running example. Note that the nodes from level i are due to shared substrings from internal nodes of level $i-1$. It is possible that a matched node (i.e., created from a match) is not a shared node (i.e., reachable from several nodes): for example, the *rad* matched node in figure 1 is not shared. Note also that nodes at level i may be created during the i-th iteration, or later.

Even if internal nodes are interesting for match visualization for a user, we select only the leaves, except the first-level interstitial leaves whose length is below the threshold (that are discarded for the following iterations), to compute similarity metrics as discussed in section 4. Edit operations breaking a longer match into smaller ones implies the creation of these discarded interstitial small leaves for the unmatched zones.

We present in figure 13 the distribution of shared functions according to the number of distinct projects that reach them (several occurrences of a function in a same project being counted as one in this classification). Then, we intentionally chose to ignore among them the leaves whose multiplicity is greater than 50



(a) Schematic view of the factorized call graph

Kind of created node	Matched internal nodes	Matched leaves	Unmatched leaves
Level 1	14512 (ml: 20.90)	1258 (ml: 10.96)	11740 inc. 1464 of length $\geq t$ (ml: 5.215)
Level 2	4219	597	13480
Level 3	874	123	3903
Level 4	200	20	832
Level 5	31	3	193
Level 6	3	0	37
Level 7	0	0	2

(b) Kind of nodes by nesting levels (with mean length of nodes for the first level)

Figure 12: Factorized call graph topology

Multiplicity (# projects) $\longrightarrow$ Length of shared nodes $\downarrow$	1	2	3..4	5..9	10..19	20..49	50..99	100..150
$[t..2t[$	1,309	2,067	1,727	1,260	524	246	60	5
$[2t..5t[$	769	1,022	629	288	83	10		
$[5t..10t[$	154	144	18	3				
$\geq 10t$	52	47	3	12				

Figure 13: Distribution of shared nodes according to their length and the number of projects in which they appear

Studied set of project pairs	Distribution by metric values					
	$[0]$	$]0..\frac{1}{10}]$	$]\frac{1}{10}..\frac{2}{10}]$	$]\frac{2}{10}..\frac{5}{10}]$	$]\frac{5}{10}..\frac{8}{10}]$	$]\frac{8}{10}..1]$
1,953 snake game pairs	209	1022	534	180	5	3
63 pairs (snake game vs. same unrelated project)	0	53	10	0	0	0

Figure 14: Distribution of pairs of projects according to the *min*-normalized similarity metric

projects, assuming they are standing for trivial code rather than for real interesting plagiarism (indeed, these leaves are shared by at least a quarter of all the projects, for different assignments). This assumption could be strengthen by studying the length of the shared leaves, or other normalized metrics presented in section 4. These 65 most-occurring leaves sums to approximately 20K tokens ($< 5\%$ of the code).

Considering only the remaining leaves, we search the shared leaves in order to quantify the amount of shared information between pairs of projects. First, we considered the global set of projects as a whole to highlight the most straightforward cases of code copy. Among the 19,701 pairs of projects, 14,909 share at least one leaf.

We examined the 26 pairs sharing more than 300 tokens for their shared leaves. Four of them share all their leaves and are exact copies (with sometimes formatting changes). For the other top pairs, a classification can be made according to the relative weights of implied projects. The pairs can be separated in two equal parts: pairs where the larger project is more than 38% more voluminous than the smaller and the other part where projects are homogeneous concerning their sizes. The data of the relative size of projects is important to assess the pattern of duplication. Among the top pairs, 3 were related to self-plagiarism cases that reuse common structures: the size of the projects were not homogeneous. Other non homogeneous pairs are due to the copy of a subset of functions from a project: in this case a high value is obtained for the *min* normalization metrics comparing the shared leaves versus the leaves of the smallest project. In all cases these top pairs reveal no false positives and similarity scores may be sometimes underestimated because of small edit operations. The copy can be due to extra- or intra-class plagiarism: in the latter situation it is not straightforward to automatically assess the direction of the copy or if the work was made in common.

Let us examine now pairs of projects from a given assignment (a snake game involving a FIFO queue). Three already examined pairs exceed the value of $\frac{8}{10}$, and 5 others are over $\frac{5}{10}$ (as reported in figure 14). We present in figure 15 two functions from copied projects placing the head of the snake on the game board (*min* normalized similarity value of their project pair: 0.62), they are highly similar by their shared leaves and the leaves they indirectly share by calling two other similar functions. Beyond the $\frac{1}{2}$ similarity threshold, projects that seem partially copied are found. They could be graphically displayed by a matrix, as shown in figure 16, where the darkness of each rectangle models the normalized metric value between two projects. Human analysis of the set of pairs of higher values confirmed the copy of code even if, in some cases, the similarity value could have been raised if identification of common sub-sequences with gaps have been made (thus handling obfuscation by tiny expression rewriting).

```
1   void InitializeWorm(int board[N][N],Worm *v){
2
3    int x,y;
4
5    do{
6     x=mlvrandom(N);
7     y=mlvrandom(N);
8    }while(board[x][y]==1||board[x][y]==2);
9    /* to not put the snake head on a prey or a trap */
10
11   InsertInHeadCell(v,x,y);
12  }
```

(a) Function from project 1

```
1   void InitNibbler(List *worm, int b[N][N]){
2
3    int x,y;
4
5    do{
6     x=random(N);
7     y=random(N);
8    }while(b[x][y]==1);
9    /* Forbid the initialization of the head on a prey */
10
11   PutInHead(worm,x,y);
12  }
```

(b) Function from project 2

Figure 15: Two highly similar functions from distinct *snake game* projects

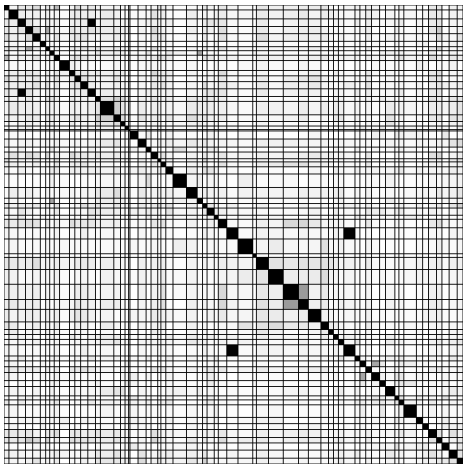


Figure 16: Matrix of *min*-normalized metric values for the snake game projects

For medium similarity scores, the examination of the similarity scores of the pairs of functions of the two involved projects may be lead. Common shared chunks of code can be represented using a slice of the factorized call graph. We reviewed some pairs scoring between $\frac{2}{10}$ and $\frac{5}{10}$: the similarity is due to the use of the same structures and idiomatic initialization, liberation and structure management code (here a matrix of positions and a FIFO structure).

Finally, we compare each project of the previous assignment to another unrelated project (a word game using a prefix tree and whose leaves summed to 1,272 tokens) that had no *a priori* reason to share something in common. The *min* normalization produces some values above $\frac{1}{10}$ for shared leaves compared to the leaves of *snake game* projects (being the smallest, up to 1,173 leaves). The most similar pair involves 131 tokens as shared leaves (51 tokens if leaves of multiplicity of at most 10 projects were ignored). Indeed, some trivial chunks of code of relatively low multiplicity could be shared like the example specified on figure 17.

```
1  for (i = 0; i < COL; i++) {
2    for (j = 0; j < ROW; j++) {
3      printf ("␣%c␣", plateau[i][j]);
```

(a) Chunk extracted from a *snake game* project

```
1  for (i=0; i < SIZE; i++) {
2    for (j=0; j < SIZE; j++)
3      printf("%c%c",d[j * SIZE + i], (j<SIZE-1)?'␣':'\n');
```

(b) Chunk extracted from the unrelated project

Figure 17: An example of a common chunk of code (project multiplicity: 3, length: 10) reported as shared

6. Related work

6.1. An insight into algorithmic approaches for similarity detection

Various approaches have already been used to search similarities in source code. Some of them chose to focus on a single project analysis while others concentrate on plagiarism detection. Often the proposed tools can be used in these two contexts. However, it intuitively appears that project clone researchers must benefit from a good precision, especially if automatic refactoring is expected. Concerning plagiarism detectors a loss of precision may be accepted but false negative matches should be avoided. However it seems unavoidable that, since we limit our scope of research to static analysis, malignant copies relying on dynamic abilities of the language (introspection, dynamic code loading...) will remain undetectable. Algorithmic equivalence detection is also out of range, being undecidable.

Another requirement concerning a source code copy detection method is its scalability. When code sites of suspected plagiarism are detected, extensive comparison with costly algorithmic methods can be envisaged. Otherwise when a large set of projects is considered the scalability of the method is essential. Unfortunately our method can only address a fixed set of projects. Conversely other techniques relies on evolving databases of fingerprints representing extracts of the code.

A first criterion to distinguish duplication detectors is their internal representation of the source code. Historically, the first tools [17] used metrics based on formatting details or token enumerations. Nowadays, these techniques appear outdated due to their high sensitivity to even the most minor edits which is detrimental if obfuscated copy is done, especially if the metrics are known to the plagiarist. The most popular representations adopted by current tools are token sequences, syntax trees, program dependency graphs and sometimes hybrid representations.

Then, considering a representation, specific abstraction and normalization steps can be applied to it. This process is easier on syntax trees where supplied syntactic information allows for example to abstract little expressions or to normalize the order of operands. A trade-off must be adopted to balance precision and recall according to the level of abstraction.

An extra step can be performed on the representation to gather elements and/or to filter them. For example tokens may be assembled to n-grams (n-tuples of consecutive tokens) themselves filtered [4] to keep only the most specific ones.

After having obtained a transformed representation, algorithmic methods should be designed to find duplications on the represented source code. For our

Method	Sequentiality	Comparison set	Memory cost[a]	Time cost	References
Element fingerprinting on k-grams	None	Indexed base	Prop. to hash size, fingerprint number	$\Theta(N \log N)$	[1, 4]
Tiling method on k-grams	Yes	Pair-wise	$\Theta(n)$	Average: $O(n^{1+\epsilon})$	[22, 23]
Suffix tree indexation	Yes	Indexed base	$\Theta(N)$	$\Theta(N)$	[20, 19]
Suffix array indexation	Yes	Fixed set	$\Theta(N)$	$\Theta(N)$	[24, 23]

Figure 18: Classical methods to search exact matches on source code

[a]Memory and time cost complexities are expressed relative to the cumulative number of tokens N for all the projects (for set comparison) or the size n of an individual project (for pair comparison).

method we try to focus on its memory efficiency. It explains the choice of a suffix array as an indexing structure rather than a suffix tree. As introduced by [18], some tools, like CCFinderX [19] or Phoenix [20], have successfully used suffix indexation structures to find duplication in source code using a tokenized form or sibling abstracted syntax sub-trees [21].

When evolution of the project database is expected, suffix tree indexation is the natural choice. Even if space-efficient suffix trees are proposed these structures remains memory-hungry. We preferred to build a new structure, the Maximal Repeated Factor (MaRF) graph, that stores only the useful repeated factors of the studied token sequences, discarding suffixes present with a single occurrence. This loss of information forbids any future use by adding new token strings. It is adapted to treat single snapshots of projects or a fixed set of projects like student assignment projects to identify internal plagiarism: any change in the studied set would imply a complete recomputation of the structures.

It clearly appears that our method can map only exactly matching sub-strings of tokens through the described suffix indexation method even if extra steps have been introduced to deal with minor edits. At our knowledge, previous tools targeted to retrieve also exact matches based on sub-strings did not address the problem of the nesting of matches. Furthermore they did not consider any function call information. In contrast taking into account the call graphs of projects to get a merged new one allows a better understanding when inlining and outlining obfuscation operations are in stake.

We propose in figure 18 a summarized classification of approaches according to their main used representation and their algorithmic method to find exact matches.

We evoke only briefly approximate match finders using dynamic alignment due to their high temporal cost: $\Theta(n^2)$ for determining edit operations on strings of n tokens to $O(n^4)$ to edit syntactic sub-trees [25]. We limit the use of such techniques to the leaves obtained after factorization (see Appendix A). Other less sensitive representations to edit obfuscation patterns like program dependency graphs (PDG) [26] could be employed: however, the algorithmic cost to determine similarity via subgraph isomorphism (a NP complete problem [27]) may be irrelevant for large set of treated data.

According to a proper abstraction and normalization of the handled representation, tools searching exact matches on token sequences may handle type 1 (differently formatted) to type 2 clones (type or identifier renaming) follow-

ing the Bellon et al. [28] and Roy et al. [29] taxonomies. If abstract syntax trees [30, 23] are considered, some type 3 to type 4 clones can even be managed (tiny expression rewriting or transposition of independent code). Except for suffix indexation techniques other approaches rely on the meta-tokenization of a string of tokens or a brotherhood of sub-trees. k-gram fingerprinting methods allow to index selected k-grams represented by hash values without consideration of sequentiality contrary to suffix indexation ones. Another popular method [22] for plagiarism detection (used by the JPlag webservice [2]) relies on progressive meta-tokenization to k-grams of variable length to cover a pair of token strings with non-overlapping clones.

6.2. Specificities of the factorization method

Our method can be compared to exact matches approaches using suffix indexation on token sequences. Relying on the same algorithmic grounds they allow the retrieval of an equivalent set of matches. The main differences are due to the various normalized representations used and to the structural view of the results (matches structure, similarity visualization...). Numerous interesting tricks could be employed to enhance the normalization of the representation: in this article we chose not to study them. We emphasize rather our work on structuring the results.

Most similarity detection tools report pairs of matching source code zones or groups (i.e. with cardinality possibly greater than 2). However nested similarities may appear in the matches themselves and generally remain unaddressed. Building the MaRF graph of token sequences allows to represent the overlapping and nesting relations between groups of exact matches. This structure is the cornerstone of the iterative factorization process by helping to the decomposition of each leaf function to shared sub-functions.

Finally our method leads to a factorized graph of functions including the original call graphs from the projects, enhanced with new synthesized functions due to similarity matching. The integration of these data permits similarity understanding at the level of the function. Pairs of functions can be compared according to their shared leaves. Most similarity detection tools limit their representation of results to entity levels such as projects, packages or compilation units. Studying similarity at a function level (potentially across several compilation units) is more rare. Godfrey and Zhou address [31] this problem in the context of origin analysis, a field connected to clone analysis and consisting in mapping entities across several versions of a same project. Their approach uses a semi-automatic iterative process based on metrics on function names, parameters and number of lines of codes. Analysis of the set of function callers and callees was used to identify refactoring patterns like splitting or merging of functions due to service consolidation/separation or elimination of clones. The factorized call graph of two versions of a project obtained by our method could also be exploited to address these concerns in origin analysis, possibly with a better accuracy and recall than techniques solely based on metrics. In a plagiarism detection context, comparing the source code at a function level can reveal similarity patterns that would be less straightforwardly highlighted otherwise. For example consider the case of a plagiarist copying a function, splitting it into several sub-functions and hiding them among unrelated code into numerous compilation units. A detection tool at a compilation unit will

reveal numerous small matches inside several functions without linking them to their caller ancestor.

7. Conclusion

In this article, we described an algorithm based on the factorization of call graphs for the detection of similarities in source code. This technique presents several advantages. It brings interesting results related to the detection of similarities in the presence of common edit operations like transposition and functional inlining and outlining. It expresses the similarities at a function level. A shared chunk of code being considered as a function (identified through the factorization process), it allows numerous duplicated chunks to be digested into a single item called by several projects. Moreover, the pre-processing cost of this approach is reasonable since, after a classical lexical analysis providing tokens, it only requires a light syntactic analysis in order to identify initial functions and their call sites. For the whole factorization process (without local alignment step), we experiment a practical time complexity that is almost linear in the number of handled tokens.

Nevertheless, this method presents two main limitations. First, this approach is not incremental. Indeed, due to the immutable nature of data structures such as suffix arrays and to the necessary choices between decompositions among overlapping shared factors, the process does not support any update of the set of analyzed projects. The second drawback is related to the fact that functions calls and related metrics are handled through sets without consideration of order. Even if we did not encounter them during experimental analysis, this can yield false positives, for instance in the case of two distinct projects involving the same set of functions. Particularly, this can occur if the factorization threshold is low: in this case, the number of leaves is low and their multiplicity high, leading to meaningless decompositions. On the contrary, a threshold that is (too) high will drastically lower the recall by raising no shared factor. Indeed, the factorization threshold must be carefully chosen to offer a good compromise between precision and recall. The choice of the threshold could benefit from statistical approach to assess the idiomaticity of code related to factors of tokens: defining new weight functions for factors considering idiomaticity metrics instead of the raw length is to be explored.

New *order* metrics can be designed to take into account the order of function calls; this could ease the refinement of the similarities previously identified with *order-free* metrics.

Providing incremental addition of projects seems difficult. To cope with this problem, the source code of analyzed projects must be indexed to be stored in a database. This is possible through fingerprinting approaches, either on tokens or in richer representations like abstract syntax trees. Abstract syntax tree representations could allow more sophisticate patterns of pre-processing of the representation for better abstraction and normalization of the code, a topic that has been neglected in this article. We are investigating some new techniques in this way [21, 32, 23] that could also consider the function call graphs of the projects.

References

[1] Moss, http://theory.stanford.edu/~aiken/moss.

[2] L. P. Prechelt, U. Karlsruhe, G. Malpohl, Finding plagiarisms among a set of programs with JPlag, Journal of Universal Computer Science 8 (2000) 1016–1038.
URL http://page.mi.fu-berlin.de/prechelt/Biblio/jplagTR.pdf

[3] M. J. Wise, Neweyes: A system for comparing biological sequences using the running karp-rabin greedy string-tiling algorithm, in: Proceedings of the 3rd International Conference on Intelligent Systems for Molecular Biology, AAAI Press, 1995, pp. 393–401.
URL http://www.it.usyd.edu.au/research/tr/tr463.pdf

[4] S. Schleimer, D. S. Wilkerson, A. Aiken, Winnowing: Local algorithms for document fingerprinting, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, ACM Press, 2003, pp. 76–85.
URL http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf

[5] R. Tarjan, Depth-first search and linear graph algorithms, in: Proceedings of the 12th Annual Symposium on Switching and Automata Theory, IEEE Computer Society, Washington, USA, 1971, pp. 114–121. doi:10.1109/SWAT.1971.10.

[6] P. Weiner, Linear pattern matching algorithm, in: 14th Annual IEEE Symposium on Switching and Automata Theory, Washington, DC, 1973, pp. 1–11.

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

[8] M. Crochemore, W. Rytter, Jewels of Stringology, World Scientific Press, 2002.

[9] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.

[10] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest common-prefix computation in suffix arrays and its applications, in: 12th Annual Symposium on Combinatorial Pattern Matching, Springer-Verlag, 2001, pp. 181–192.

[11] E. M. McCreight, A space-economical suffix tree construction algorithm 23 (2) (1976) 262–272.

[12] E. Ukkonen, Constructing suffix trees on-line in linear time, in: J. van Leeuwen (Ed.), 12th, Madrid, Spain, 1992, pp. 484–492.
URL http://cs.helsinki.fi/u/ukkonen/SuffixT1.ps

[13] S. Kurtz, Reducing the space requirement of suffix trees, Software - Practice and Experience 29 (1998) 1149–1171.

[14] M. I. Abouelhoda, S. Kurtz, E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, Journal of Discrete Algorithms.
URL http://www.fli-leibniz.de/www_bioc/journal_club/AboKurOhl2004.pdf

[15] O. Berkman, U. Vishkin, Recursive star-tree parallel data structure, SIAM Journal on Computing 22 (2) (1993) 221–242. doi:10.1137/0222017.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, second edition, MIT Press and McGraw-Hill, 2001.

[17] K. J. Ottenstein, An algorithmic approach to the detection and prevention of plagiarism, SIGCSE Bulletin 8 (4) (1976) 30–41. doi:10.1145/382222.382462.

[18] B. S. Baker, A theory of parameterized pattern matching: algorithms and applications, in: Proceedings of the 25th annual ACM symposium on Theory of computing, ACM, New York, USA, 1993, pp. 71–80. doi:10.1145/167088.167115.

[19] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: A multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (7) (2002) 654–670. doi:10.1109/TSE.2002.1019480.

[20] R. Tairas, J. Gray, Phoenix-based clone detection using suffix trees, in: Proceedings of the 44th annual Southeast regional conference, ACM, New York, USA, 2006, pp. 679–684.
URL http://www.cis.uab.edu/gray/Pubs/acmse-2006-robert.pdf

[21] M. Chilowicz, É. Duris, G. Roussel, Syntax tree fingerprinting for source code similarity detection, in: 17th IEEE International Conference on Program Comprehension, IEEE Computer Society, Vancouver, BC, Canada, 2009, pp. 243–247. doi:10.1109/ICPC.2009.5090050.

[22] M. Wise, String similarity via Greedy String Tiling and Running Karp--Rabin matching, Tech. rep., Dept. of CS, University of Sydney (1993).
URL http://www.pam1.bcs.uwa.edu.au/~michaelw/ftp/doc/RKR_GST.ps

[23] M. Chilowicz, Recherche de similarité dans du code source, Ph.D. thesis (2010).
URL http://igm.univ-mlv.fr/~chilowi/research/phd/

[24] M. Chilowicz, E. Duris, G. Roussel, Finding similarities in source code through factorization, in: A. Johnstone, J. Vinju (Eds.), 8th Workshop on Language Descriptions, Tools and Applications, Vol. 238 of Electronic Notes in Theoretical Computer Science, Elsevier, Budapest, Hungary, 2008, pp. 47–62, (15 pp.). doi:10.1016/j.entcs.2009.09.040.

[25] K. Zhang, D. Shasha, Simple fast algorithms for the editing distance between trees and related problems, SIAM Journal of Computing 18 (6) (1989) 1245–1262. doi:10.1137/0218082.

[26] J. Krinke, Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, 2001. URL http://www.bauhaus-stuttgart.de/clones/ast01.pdf

[27] S. A. Cook, The complexity of theorem-proving procedures, in: Proceedings of the third annual ACM symposium on Theory of computing, ACM, New York, USA, 1971, pp. 151–158. doi:10.1145/800157.805047.

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

[29] C. K. Roy, J. R. Cordy, Scenario-based comparison of clone detection techniques, in: Proceedings of the 16th International Conference on Program Comprehension, IEEE, 2008, pp. 153–162.

[30] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Washington, USA, 1998, p. 368. doi:10.1109/ICSM.1998.738528.

[31] M. Godfrey, L. Zou, Using origin analysis to detect merging and splitting of source code entities, IEEE Transactions on Software Engineering (2005) 166–181.

[32] M. Chilowicz, É. Duris, G. Roussel, Syntax tree fingerprinting: a foundation for source code similarity detection, Tech. rep., LIGM, Université Paris-Est (2009).
URL http://igm.univ-mlv.fr/LIGM/internal_report/pdf/2009_03.pdf

Appendix A. Dealing with local edits

In section 2.3, we noticed with the help of an example a general problem linked to all exact factor matching approaches. This problem is related to the presence of tiny local transforms (additions, deletions and substitutions) involving mostly expression rewriting. Thus the flow of exact matching tokens is broken: a duplicated factor of length above the threshold can be broken into undetectable small factors that individually do not reach the detection threshold. An obfuscator could exploit this weakness introducing neutral tokens (e.g. multiplying an integer by one...) that are not always removable through a statical normalization process. In this appendix, we briefly explore a way to cope with this problem. The main idea consists in looking for nearly similar leaves obtained after factorization using a dynamic programming alignment approach. Indeed, only dynamic programming alignment approach addresses the problem by looking for sub-sequence duplication (allowing gaps) rather than sub-string (factor) duplication. However, their complexity is prohibitive for the analysis of large projects or sets of projects. This is why we use heuristics in order to only select interesting leaves for further examination.

To allow factorization by common sub-sequences and not only by shared sub-strings, we introduce after the last factorization iteration some new sub-sequence factorization. Instead of using a suffix array to identify shared factors among the leaves, we compare each pair of (interstitial or not) leaves above the length threshold using a Smith–Waterman like local alignment method in average time complexity of $O(\beta^2 k^2)$ to compare β leaves of mean length k. This approach appears computationally more efficient than comparing token sequences from complete projects since we work at a function level, avoiding the overhead of aligning several times the shared factors and of comparing the synthesized interstitial leaves under detection threshold t. However, for projects with few leaves of small duplicity (especially interstitial leaves) the time complexity is hardly improved. When interesting sub-sequences (above the weight threshold t with limited token gaps) are reported they are factorized synthesizing in a single new function leaf. Therefore functions that were considered non similar before the sub-sequence factorization because they were sharing few leaves, may appear similar after this step if they reach leaves that are nearly identical.

Due to the detrimental time complexity relative to the alignment of all of the pairs of leaves, it is important to highlight the pairs that are more prone to host interesting sub-sequences. For this task we can assume that sequences sharing multiple tiny factors (determined via suffix indexation with a suffix array [23]) should be candidate to a local alignment process.

Appendix B. Metrics comparison tests

In this appendix, we try to compare the (*min* normalized) metrics obtained via our factorization method to metrics proposed by other popular plagiarism detection tools, according to distinct obfuscation techniques. Unfortunately, probably to deter plagiarists, most of existing tools are available as web services and are not distributed, neither under a closed nor under an open source form. Among them, we selected JPlag and Moss. JPlag [2] advertises itself as using a meta-tokenization process on pairs of projects (the Running Karp–Rabin Greedy String Tiling algorithm): its goal is to maximize the length of matches of exact shared factors between two projects and to avoid overlapping between them. Moss [1] rather uses a token k-gram hashing approach with a selection of the generated fingerprints using the winnowing method [4]. We note that these two tools seem to compare a set of projects through individual comparison of their pairs rather than relying on a global process (for the whole set of projects) like our factorization method. Furthermore their internal representation of the similarities differs: independent matches of similar chunks are reported without consideration of nested similarities and function call links. Finding an edge of comparison is not straightforward: we choose a restricted and unperfect criterion based on *min*-normalized similarity scores between projects. The aim is to assess the practical usefulness of our similarity metrics infered from the shared factorized leaves rather than doing a raw efficiency comparison of different tools.

We selected a modest sized student assignment (mean of 600 lines of ANSI C code) with 36 submissions and applied on them a raw tokenization and light syntactic analysis to extract functions and call tokens without further normalization process. Among the projects one project was chosen to be humanly obfuscated using various methods:

1. *Raw duplication.* No modification was made on the code.
2. *Identifier substitutions.* The local variable names were replaced by random names.
3. *Transposition of functions.* The functions were permuted inside a same source compilation unit.
4. *Regular insertions and deletions of instructions.* Small instructions were regularly added in the source code whereas instructions appearing as useless were removed.
5. *Inlining of functions.* All the function calls were replaced by their body up to a single level.
6. *Outlining of parts functions.* Some function chunks were moved outside their parent function and replaced by a function call.
7. *Useless code flooding.* Large blocks of useless code were added in the original project to dilute the clones.
8. *Non plagiarized.* An other project containing no code duplication of the original project according to a human judge was added to assess false positivity report of the tested tools. Contrary to tests in 5.2 we did not discard high multiplicity leaves to compute the similarity scores.

As for the factorization method, JPlag match report threshold was set to 10 tokens whereas pattern matching parameters for Moss were not known. All of the tools used an min-normalized normalization technique measuring the shared amount of code versus the smallest project. Results are expressed in figure B.19

Kind of obfuscation	1	2	3	4	5	6	7	8
Factorization	1.0	1.0	1.0	0.72	0.87	0.81	0.84	0.04
Moss	0.94	0.94	0.90	0.25	0.74	0.56	0.73	0.01
JPlag	0.99	0.99	0.97	0.37	0.87	0.86	0.81	0.00

Figure B.19: *Min*-normalized similarity scores between an original project and hand-crafted obfuscated versions

for the computed similarity metrics between the original project and the obfuscated versions. Ideally they should reach 1.0 for all obfuscation techniques (except the 8th non plagiarized version). It is the case for exact cloning, identifier substitutions and function transposition for the factorization method that are call graph-conservative. It is surprisingly not the case for Moss maybe due to a normalization based on the complete source code without comment discarding. For our tested factorization method results are homogeneous for other obfuscation schemes. We especially note the interesting recall for the code insertion and deletion pattern even if long similarities are unavoidably broken into smaller ones that may be winnowed due to the token threshold.