Language-Independent Clone Detection Applied to Plagiarism Detection

Romain Brixtel, Mathieu Fontaine, Boris Lesner, Cyril Bazin
GREYC-CNRS (UMR-6072)
University of Caen Basse-Normandie
14000 Caen, France
Email : {firstname}.{lastname}@info.unicaen.fr

Romain Robbes
DCC, University of Chile
Blanco Encalada 2120, Off. 308
837-0459 Santiago, Chile
Email : rrobbes@dcc.uchile.cl

***Abstract*—Clone detection is usually applied in the context of detecting small-to medium scale fragments of duplicated code in large software systems. In this paper, we address the problem of clone detection applied to plagiarism detection in the context of source code assignments done by computer science students. Plagiarism detection comes with a distinct set of constraints to usual clone detection approaches, which influenced the design of the approach we present in this paper. For instance, the source code can be heavily changed at a superficial level (in an attempt to look genuine), yet be functionally very similar.**

Since assignments turned in by computer science students can be in a variety of languages, we work at the syntactic level and do not consider the source-code semantics. Consequently, the approach we propose is endogenous and makes no assumption about the programming language being analysed. It is based on an alignment method using the parallel principle at local resolution (character level) to compute similarities between documents. We tested our framework on hundreds of real source files, involving a wide array of programming languages (Java, C, Python, PHP, Haskell, bash). Our approach allowed us to discover previously undetected frauds, and to empirically evaluate its accuracy and robustness.

***Keywords*-Endogenous; Plagiarism Detection; Similarity Measure; Distance; Source Code Segmentation; Source Code Plagiarism**

I. Introduction

As computer science teachers we sometimes have to deal with unethical students who copy other's work (ie. source code) for their projects. Tracking this plagiarism is a time-consuming task since it requires comparing each pair of documents containing hundreds or even thousands of lines. In such a context, the idea to apply clone detection approaches to the problem of plagiarism detection is appealing.

However, if clone detection and plagiarism detection share the same goal—to detect highly similar source code fragments—they operate under a very different set of assumptions. A typical code clone is often created because the original source code could not be reused as-is. Hence, behavioral modifications are rather common [1].

Plagiarised code strives to keep a behavior as close to the original as possible, while actively trying to avoid detection, by renaming variables, altering the layout of the code, changing code comments, *etc.*

A further complicating factor in the case of plagiarism detection is that the process must necessarily be lightweight. If an assignment happens to be in a programming language that the approach does not support, the effort to adapt it is greater than the time required to look for plagiarism manually, hence the approach is useless. Language-independent approaches are thus to be preferred.

This paper introduces a language-independent code duplication detection approach geared towards source code plagiarism. We applied it on several corpora, consisting of several hundreds of source code files, in a variety of programming languages.

Contributions. The contributions of this paper are:

- An analysis of the different assumptions one must take between classical code clone and plagiarism detection.
- The detailed presentation of our plagiarism detection approach which is comprised of document segmentation, similarity measurements, and report generation.
- An empirical evaluation of its accuracy on a corpus of several hundred documents.

Structure of the paper. Section II presents related work on clone detection, and plagiarism detection. Section III outlines the differences between plagiarism and clone detection approaches, and presents the model of source code plagiarism that our approach is based on. Section IV gives a detailed account of our approach, while Section V presents the empirical results we obtained on a corpus containing several languages. Finally, Section VI discusses our approach, and Section VII both concludes and outlines future work.

II. Related Work

In this section, we review clone detection approaches—focussing on language-independent ones—before reviewing dedicated plagiarism detection approaches.

A. Clone Detection

Clone detection is a long-lasting problem in the software maintenance field, that has spawned a number of approaches. Approaches can be language-specific, such as the one of Baxter *et al.*, based on comparing abstract syntax trees [2], or the one of Krinke, based on dependence graphs [3]. In the following, we focus on language-independent (or

more easily adaptable) approaches, as plagiarism detection approaches should be lightweight enough to be applicable on a variety of languages.

Baker presented a line-based approach that is able to handle systematic renaming of variables [4]. Baker's tool, *Dup* only needs to perform lexical analysis to detect variable substitutions, hence it can be adapted to other languages with a moderate cost (it was validated on C code).

Kamiya *et al.* introduced CCFinder, another token-based tool [5]. CCFinder can find clones in C, C++, Java, and COBOL. The authors mention that adapting the tool to Java took two days, and to COBOL, one week. CCFinder handles variable substitutions in order to find clones where some variables were renamed.

Ducasse *et al.* introduced an approach based on string matching, which uses a lightweight method to be adapted to new languages [6]. The authors mentioned that the time to incorporate a new language was in all cases less than 45 minutes. The approach handles several code normalisation schemes, in order to cope with renaming of variables, constants and function names. The author found that excessive normalisation was harmful to the precision of the approach.

Wettel and Marinescu introduced an approach aimed at discovering larger code clone fragments by finding duplication chains in the source code, *i.e.,* sequences of smaller code clones [7]. Unfortunately, the approach does not handle renames of variables. It is however truly language-independent: only an optional feature (insensitivity to comments) is language-specific.

Finally, Bellon *et al.* performed a through evaluation of code clone detection tools on the same corpus of data [8]. The clones were classified in three categories: identical clones (type-1), Clones with substitutions (type-2), and clones with further modifications (type-3).

B. Plagiarism Detection

Many plagiarism detection frameworks exist. Plague [9], YAP3 [10] and Jplag [11] [12] decompose the source-code into a sequence of lexemes (the lexemes are variables, attributes, methods, etc.). This sequence is used as a pivot language to compare the similarity between the documents. In the same way, MOSS [13] uses an n-gram model to compute a fingerprint of each document. MOSS assumes that plagiarised documents have close fingerprints. All these methods require *a priori* knowledge like a list of the language keywords or comment detection rules.

Anti-Copias [14] exploits two kind of similarities. The first one considers a document as a repartition in a vector-space of tokens. The idea is that two plagiarised documents, sharing a significant amount of code, are described by a similar vector. The other one uses the information distance to compute the similarity between each pair of documents in the corpus. Given two documents, it computes an approximation of the information distance using a compression algorithm (see section IV-C for details). This approximated distance is easy and fast to compute as it only requires a compression algorithm (zip, rar, lzh, etc.). Those distances have no assumptions on the positions of similar source-code portion across assignments. Anti-Copias considers fraud to be exceptional, thus highlighting documents that are found to be very similar wrt. the rest of the corpus.

Cosma's approach is based on language similarity, but handles the problem of renaming through the use of Latent Semantic Analysis, which extracts topics based on occurences of words together in the source code, and is as such less sensitive to synonymy, polysemy and renamings in general [15].

Son *et al.* use a large amount of source code information and compare parse trees to detect plagiarism in order to handle cases where a large amount of uninterpreted code is added [16].

III. Specificities of source code plagiarism detection

A. Differing Assumptions

Having seen a variety of approaches in both clone and plagiarism detection, we can highlight the main differences between the two kind of approaches.

Amount and kinds of transformations. How (and how much) source code is transformed before being submitted varies greatly. Plagiarised code usually features a large amount of shallow transformation, in order to make the fraud undetectable at a glance. On the other hand, the code must behave similarly to the copied solution, as the student assumes that it is correct. In the classification found in Bellon's study, this correspond to type-2 clones [8], although the renaming might be more extensive in case of plagiarised code. The amount of renaming involved in plagiarised code may contradict the results found by Ducasse *et al.* [6], stating that excessive normalisation of the source code might be an issue: "excessive" normalisation might be the only way to find some heavily edited code clones.

Code clones found in software systems are often the results of code that can not be conveniently abstracted. Hence the programmer might have a greater tendency to either leave them unaltered (type-1 clones), or to modify their behavior to adapt them to the case at hand (type-3 clones). Type-1 clones are inherently risky for students, while advanced plagiarisers may attempt to further change the code and produce type-3 clones (for example by reordering statements). There has been some studies about type-1 [17] and type-3 clones [18] in software.

Size of the duplication. Clone fragments tend to be typically small; plagiarised code on the contrary often involves large scale reuse of the original code. Clone detection approaches tend to return specific portions of the source-code that are believed to be duplicated (some, such as Wettel and Marinescu, would argue that they are too small

and that the fragments should be grouped [7]). Plagiarism on the other hand is often decided at the document level. Either the whole (or significant portions) are plagiarised, or the document is genuine. This gives plagiarism detection approaches a larger amount of data before deciding which is which.

Time constraints. Duplicated code is often an issue in large-scale software systems. Hence, the performance of clone detection is a significant concern. Recent contributions have been focused on parallelising the clone detection process in order to apply it to very large code bases [19]. Detecting plagiarism is on the other hand done at a smaller scale, allowing more time-consuming algorithms to be more practical.

Amount of results. Clone detection on large-scale systems means a lot of clone candidates. Detecting plagiarism in assignments is however a smaller-scale endeavour. Hence the general requirements in terms of balance between precision and recall may vary between the approaches. If in general a good recall is desired, a decent amount of precision is also important in clone detection, in order to avoid looking at thousands of irrelevant clones. In the case of plagiarism, a high recall is even more important, even if precision has to suffer somewhat.

Language-independence. In order to deal with the variety of programming languages that students might have to write in during their studies, a language-independent approach is desired, in order to minimise the amount of time spent in adapting the approach to new languages. Of the clone detection approaches we surveyed, only one (the approach of Wettel and Marinescu) was truly language independent. All the others needed an adaptation phase to work with a new language. This phase can take from dozens of minutes (in the case of Ducasse *et al.*), up to days, or weeks (for Kamiya *et al.*). This investment is often too heavy for assignment verification.

B. A model of plagiarism

In source-code documents, we define plagiarism as the application of successive *transformations* applied on an original document. A transformation only modifies the structure and appearance of the source code, but not the program's functionality. We define four kinds of transformations with respect to which our method must be robust:

Renaming: This is a basic transformation where identifiers (variables, function names, *etc.*) are renamed. It can be done very easily with current development suites or even text editors.

Code reordering: This consists in moving pieces of source-code inside the document, such as functions, variable declarations, *etc.* This is an easy transformation to achieve in practice.

Using uninterpreted text or dead code: The most obvious example is adding or removing comments or dead code as well as indenting or using blank lines, that is, anything not being interpreted by the code compiler.

Equivalent structures: This is the hardest transformation to achieve, as it requires using different code structures or instructions to behave similarly at execution time: a basic example is replacing a `for` loop with a `while` loop. Beyond some (subjective) point, this transformation may not be considered plagiarism anymore.

We consider the more transformations are done, the less two documents are plagiarised. If a student can achieve all these transformations a sufficient amount of times, we will not be able to find the plagiarism. In this case, it could have cost him more time than required to write original work.

IV. Detecting Plagiarism

A. Objectives

We built a framework to help teachers finding plagiarised documents into a corpus of source-code files. We consider two or more documents to be *suspects* if they are much more similar than the average similarity between documents. Sometimes students copy part of their work from external resources (e.g., the Internet). In such a case, their work is very different from the rest of the documents. We call these documents *"exceptional"*. These documents are either plagiarised from external resources, or written by great students; only teachers can remove the ambiguity.

Our framework builds a report highlighting the groups of suspicious documents and the exceptional ones. The teacher can hence investigate plagiarism on restricted subsets of the corpus.

Our plagiarism detection approach comprises six stages (figure 1):

1) pre-filtering,
2) segmentation and similarity measurement stage,
3) segment matching,
4) post-filtering,
5) document-wise distance evaluation,
6) and corpus analysis presentation.

This is a bottom-up approach, in the sense that it first operates at a character level (stages 1 and 2), then subsequently at the segment (string) level (stages 2, 3 , 4, and 5), the document level (stages 5 and 6) and finally the corpus level (stage 6).

Our framework considers that two documents are plagiarised if it detects an abnormal amount of similar consecutive segments in those documents. The computation of consecutive segment similarity is achieved in stage 2, 3, 4 and 5.

In this context, our framework takes care of all the transformations presented in section III. Stage 1 aims to takes care of variable renaming. We make the assumption that code reordering consists in moving a group of consecutive segments, thus this transformation doesn't alter the detection. Considering that the use of equivalent structure

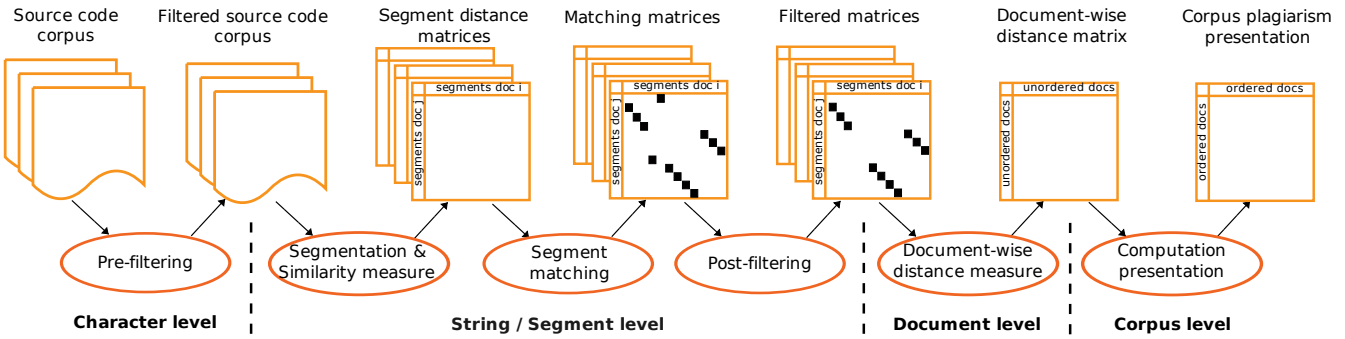


Figure 1. Overview of the process pipeline

will neither affect too much the segments composing the structure nor their order, such a transformation doesn't greatly compromises the detection. For example replacing a `for` loop by a `while` loop will not affect the content of the loop. Finally, the use of uninterpreted text is either a local modification of the segment (taken care in step 1 and 2) or the addition of segments which should not modify too much the detection of groups of similar consecutive segments.

Algorithm 1 describes how our plagiarism detection operates. Similarity measures in stage 2 and subsequent stages 3, 4, and 5, form the procedure DOCUMENTDISTANCE (algorithm 2), while stage 6 is performed in the procedure DISPLAY whose operation is described in section IV-G. Figure 1 shows the entire pipeline and the data entering and exiting each stage. The following subsections will present the six stages and their relationships.

Algorithm 1: Main algorithm

Input: $\mathcal{D}$: a set of documents
Data: $PreFilter$: a pre-filtering function
Data: Seg : a segmentation function
Data: $Dist$: a segment distance function
begin
  /* Pre-filter the documents, CAN BE PARALLELISED */
  $\mathcal{D}' \leftarrow \{PreFilter(d) \mid d \in \mathcal{D}\}$
  /* Split each filtered document into a set of segments, CAN BE PARALLELISED */
  foreach $d'_i \in \mathcal{D}'$ **do** $S_i \leftarrow Seg(d'_i)$
  /* Compute the distance between each pair of documents, CAN BE PARALLELISED */
  foreach $d'_i, d'_j \in \mathcal{D}',\ i > j$ **do**
    $\mathcal{M}_{(i,j)} \leftarrow$ DOCUMENTDISTANCE ($Dist$, S_i, S_j)
  /* Return a human-readable result */
  return DISPLAY ($\mathcal{M}$)
end

B. Pre-filtering

The first stage makes the detection process robust to the first transformation : *renaming*. For that, we rename each token (in many languages an alphanumeric string with underscores) by a single symbol (Figure 2). This is inspired by the work of Urvoy *et al.* [20] on web spam detection. As mentioned above, the work of Ducasse *et al.* found that excessive normalisation of the source code is harmful for the precision of clone detection, but this assumption does not hold for plagiarism detection.

C. Segmentation and similarity measure

In the second stage, each document is divided into segments. Our approach deals with *code reordering* by detecting similar segments between two documents. Therefore, we work at different granularity levels: finer than the whole document and coarser than characters. Intuitively, the segmentation determines what will be a "unit" of code. For example, we may want to work at the line level or the function level.

We now introduce some notations and properties about segments. Let Σ be an alphabet. A *document* is an element of Σ^*. For any character string $s \in \Sigma^*$, we write $s[i]$ for the i^{th} character of s, and $s[i,j]$ (with $i \leq j$) is the substring $s[i]s[i+1]\dots s[j]$.

A *segment* is a contiguous subset of a document. More formally, a segment formed on a document d is an element $(d,p,l) \in \Sigma^* \times \mathbb{N}^* \times \mathbb{N}^*$ where p and l are respectively the starting position and the length of the segment.

A *segmentation function* Seg partitions a document d into a sequence of segments $Seg(d) = (s_1, \dots, s_m)$ such that the segments are contiguous and the text of their concatenation is equal to d.

For example, a segmentation function can split the document at line breaks. In this case, a segment is a line of pre-filtered source code and we try to detect an abnormal number of similar consecutive lines in two documents. The similarity

Original source-code

```
char ** cut(char *str) {
  char d[] = " ";
  char **result= NULL;
  int i = 0;
  int s = 10*sizeof(char *[20])
  res = (char **) malloc(s);
  res[0] = strtok(str, d);

  while (result[i] != NULL) {
    i++;
    res[i] = strtok(NULL, d);
  }
  return res;
}
```

Pre-filtered source-code

```
t ** t(t *t) {
  t t[] = " ";
  t **t= t;
  t t = t;
  t t = t*t(t *[t])
  t = (t **) t(t);
  t[t] = t(t, t);

  t (t[i] != t) {
    t++;
    t[t] = t(t, t);
  }
  t t;
}
```

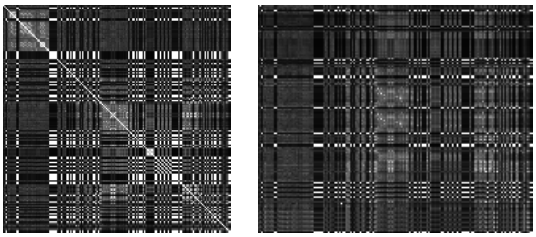
Figure 2. Sample of a source-code before and after pre-filtering

between two segments is given by a distance function.

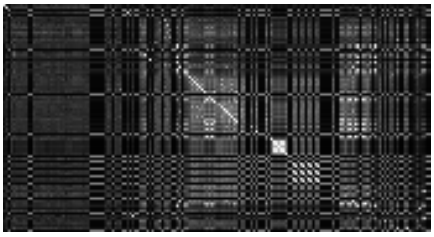
A distance function $Dist(s_1, s_2)$ between two segments returns a real number in $[0, 1]$ satisfying the usual distance properties.

For two given segmentations $S_1 = (s_1^1, \ldots, s_m^1)$ and $S_2 = (s_1^2, \ldots, s_n^2)$ and a distance function $Dist$, we obtain a $m \times n$ *distance matrix* $\mathcal{M}$ where $\mathcal{M}_{(i,j)} = Dist(s_i^1, s_j^2)$.

Figure 3 depicts distance matrices from samples of three kinds of documents. Note that contiguous and similar sections of documents make diagonals appear like in Figure 3(a) and less obviously in Figure 3(c) where only parts of documents are plagiarised. It's important to see that dissimilar documents have no diagonals in their segments distance matrix.



(a) identical documents (b) different documents



(c) plagiarised documents

Figure 3. Distance matrices $\mathcal{M}$ for three pairs of documents, obtained by segmenting the document into lines and using the Levenshtein distance. A pixel represents the distance between two segments. The lighter the point, the smaller the distance.

Any distance function can be used to compare segments, such as the Hamming or Levenshtein [21] (aka. edit) distance counting the number of operations (inserts, deletes, or replaces) to transform one segment into another. Another interesting distance is the information distance; despite being uncomputable, it can be approximated using data compression [22]. Let c be a compressor and $|c(s)|$ be the size of the compressed version of s using c, thus the distance between two strings can be expressed as

$$Dist(s_1, s_2) = 1 - \frac{|c(s_1)| + |c(s_2)| - |c(s_1, s_2)|}{\max(|c(s_1)|, |c(s_2)|)}$$

Due to the metadata produced by a compressor such as `gzip`, such a distance should not be used on short segments (*e.g.,* lines) since the size overhead induced by metadata is not negligible wrt. the total compressed size. The information distance is used in other plagiarism detectors such as Anti-Copias [14] to compute a document-wise distance.

D. Segment matching

At this point, we have a distance matrix $\mathcal{M}$ for a pair of segmentations S_1 and S_2. From such a matrix, we want to find a distance between documents themselves. To that aim, we look for a maximal matching [1] of minimal distance between the segments of both documents. A matching is a set of pairs $C \subset S_1 \times S_2$, such that each segment of S_1 and S_2 appears in at most one pair of C. A matching C is *maximal* iff all the segments of the smallest segmentation are in C, and therefore $|C| = \min(|S_1|, |S_2|)$. The *distance* of a matching C is defined as : $\sum_{(s_i^1, s_j^2) \in C} \mathcal{M}_{(i,j)}$.

This stage makes the method robust to uninterpreted text modifications: if some comments are added, the size of the document will be greater than the original one, making the segments corresponding to this new text unlikely to be matched with the original segments. Even if this new uninterpreted text matches, it can be handled in the next stage: filtering.

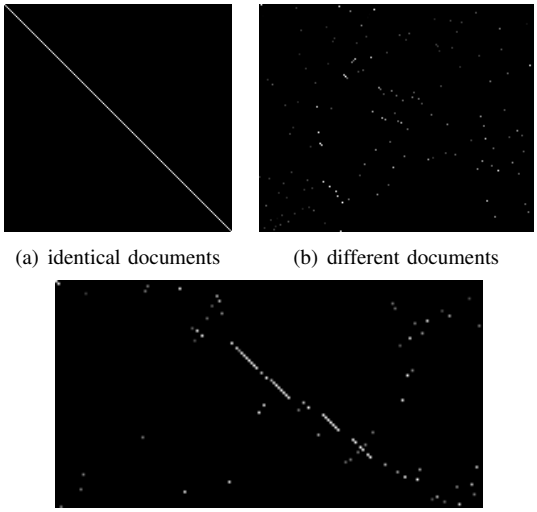
We use the Munkres algorithm [23] to perform the matching and obtain a $m \times n$ *matching matrix* $\mathcal{H}$ such that $\mathcal{H}_{(i,j)} = \mathcal{M}_{(i,j)}$ if $(s_i^1, s_j^2) \in C$ and $\mathcal{H}_{(i,j)} = 1$ otherwise. This algorithm performs in $O(\max(m, n)^3)$ time.

In Figure 4, we have the matching matrices produced by the Munkres algorithm on the distance matrices previously shown in Figure 3. In these figures we can see how the diagonals are emphasised, as well as how much the noise is reduced.

E. Post-Filtering

This stage comes from the previously stated observation that similar documents will often have consecutive segments paired by the matching stage, yielding diagonals of elements on the matrix containing values lesser than 1. Dissimilar documents will mostly have isolated points with values close to 1.

[1] The matching may not be complete since the segmentations may be of a different size



(a) identical documents (b) different documents

(c) plagiarised documents

Figure 4. Matching matrices $\mathcal{H}$ from the distance matrices of Figure 3.

Algorithm 2: DOCUMENTDISTANCE

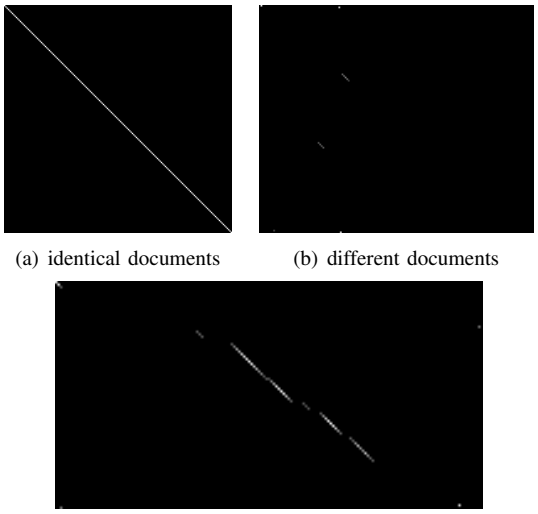
Input: $Dist$: a segment distance function
Input: S_1, S_2: two sets of segments (one per document)
Data: $PostFilter$: A post filtering function
Data: $Matcher$: A matching function
begin
  /* Build the segments distance matrix $\mathcal{M}$ */
  foreach $(s_i, s_j) \in S_1 \times S_2$ **do**
    $\mathcal{M}_{(i,j)} \leftarrow Dist(s_i, s_j)$
  /* Find the maximal segment matching, with minimal distance */
  $\mathcal{H} \leftarrow Matcher(\mathcal{M})$
  /* Post-filter the matching matrix */
  $\mathcal{P} \leftarrow PostFilter(\mathcal{H})$
  /* Return the document-wise distance */
  return $1 - \frac{1}{\min(|S_1|,|S_2|)} \sum_{i,j} 1 - \mathcal{P}_{(i,j)}$
end

This observation echoes those of Veronis [24] highlights on paragraph and sentences alignment problem. This problem consists of finding, in a text and its translations, equivalent passages in a semantic way. When the sentence or paragraph level are considered, alignment methods use the parallel criterion which consists in two main assumptions :

- Quasi-monotonous : the order of the sentences are the same or very close ;
- Quasi-bijectivity : the large majority of alignments are $1:1$ (one sentence matches only another one), or the few $m:n$ alignments that do exists are limited to small m and n values (usually ≤ 2).

Back to the plagiarism detection, we assume that blocks of instructions (for example: functions or methods) can move without efforts across a source code but the instructions inside those blocks are constrainted by the parallel criterion. In order to exploit those assumptions, we use a convolution matrix followed by a filter so that consecutive matched segments are emphasised and isolated (poor) matches are removed. This leads to the post-filtered matrix $\mathcal{P}$.

We filter the matching matrix using a smaller *identity convolution matrix*. A second filtering step is *thresholding* where every element of the matrix (after convolution) is greater than a threshold (empirically set to 0.7). Figure 5 depicts the effect of these filters on the previous matching matrices from Figure 4. In these matrices, non-contiguous matches are removed and diagonals are enhanced.



(a) identical documents (b) different documents

(c) plagiarised documents

Figure 5. Filtered matrices $\mathcal{P}$ of segments matches from figure 4, with a 5×5 convolution matrix and a 0.7 threshold.

F. Document-wise distance evaluation

From a filtered matching matrix $\mathcal{P}$ between two segmentations built from two documents d_1 and d_2 we can compute a document distance $\delta(d_1, d_2)$ defined as:

$$\delta(d_1, d_2) = 1 - \frac{1}{min(|S_1|, |S_2|)} \sum_{i,j} 1 - \mathcal{P}_{(i,j)}$$

We sum and normalise the matrix giving a distance between two documents in the range $[0, 1]$. Intuitively, the more we see diagonals in matrix $\mathcal{P}$, the closer the documents are.

G. Corpus analysis presentation

At this point we have a document-wise distance for every pair of documents in the corpus. We put these distances in a

spreadsheet where each cell contains a distance and is emphasised by a color (see Figures 6 and 7 for examples). The cell color represents the similarity between documents wrt. the average corpus similarity. The documents are ordered in a way that displays close documents (in terms of similarity measure) in neighboured cells.

For coloring, we used *nested means* to classify pairs of documents into 8 classes, each one colored on a scale going from green (legitimate documents) to red (probable plagiarism). Note that the number of classes can be extended for large corpora.

In order to bring similar documents together, we use a hierarchical classification algorithm to build a dendrogram (a binary tree) whose leaves are the documents. This dendrogram is then traversed in a depth-first fashion giving an order on the documents. To build the dendrogram, we first construct a leaf node for each document. Then we find the two nodes with minimal distance: the distance between two nodes n_i, n_j being the maximal distance between a leaf (document) of n_i and a leaf of n_j. Next, we construct a new node having n_i and n_j as children. This process is repeated until we obtain a dendrogram of the corpus.

For the depth-first traversal, at each node the child with the greatest number of leaves is visited first. The final scores table will display the documents in the order they are visited.

V. Experiments and results interpretation

All of our experiments were conducted on source-code submitted by computer science students as programming homework, without them knowing that their code would be checked for plagiarism. We tested different pipeline settings: with or without pre-filtering, many segmentation functions (*n-grams*, items may be lines, characters, *etc.*), different segment distances (Levenshtein distance or approximated information distance), *etc.* Even if the comparison of different pipelines is a very interesting experiment by itself, in this paper we chose to demonstrate the good results obtained using one of these pipelines only. We used the following pipeline configuration: tokens pre-filtering, line-by-line segmentation, Levenshtein segment distance, 5×5 convolution matrix and 0.7 threshold for post filtering. This configuration was used on the experiments presented in this paper and seems to be reliable on files having less than 1000 lines. For larger files, we also worked on a segmentation algorithm using *maximal repeats* in strings to have segments corresponding to code blocks (functions, loop bodies, *etc.*) but this remains to be fully evaluated.

As shown on Figures 6 and 7, only a few pairs of documents are displayed as "suspicious" thus greatly reducing the number of documents to be checked manually. We present here the results obtained by processing two corpora, but our method was also tested (and validated) on hundreds of real students source files, with some corpora having over 100 files, in many different programming languages such as C, Python, Haskell, Bash, PHP and Java.

The Haskell corpus on Figure 6 contains 13 documents of about 400 lines each; plagiarism detection took around five minutes. We can clearly see that the pairs (2,1), (5,3) and (12, 11) are suspects. The filtered segments distance matrices allow us to find how these documents are plagiarised. Indeed, documents 5 and 3 are virtually identical, except a contiguous part being moved in the copy. After a manual check, even if documents 11 and 12 seem similar it is difficult to decide if one inspired the other. Documents 1 and 2 on the contrary are very different from the rest of the corpus. Document 1 was taken from the Internet: the source code and comments are written respectively in English and Japanese. Document 2 has been produced by a student using successive modifications of this source-code (matching our model). When document 1 was not present in the corpus, we still were able to suspect document 2 due to its high dissimilarity to the others. Our software found the suspicious documents which turned out to be plagiarised, and also highlights document 7 which was written by a very good student (exceptional by our definition).

The second test corpus, written in Python, whose results are shown in Figure 7, has 15 files of about 150 lines of code each; computation took less than a minute. On this figure we can see two groups of plagiarised files : (5,2) and (1,7,13). Document 14 was a common code base given to all students. Note how it is similar to many others, except where students rewrote everything from scratch (documents 4, 9, 12).

In the third corpus, students had to create a tiny client/server architecture to execute shell commands. The source code given by the students are divided in two files: "client.c" and "server.c". In order to use our plagiarism detection tool, we concatenate both files, considering that blocks of codes can be moved from one to the other. The average size of a project is about 250 code lines. According to the Figure 8, the couple (4,31) is suspect. When reading the files we happened to see that the client files are pretty much the same, only indentation and some line breaks were modified. The server files are also plagiarised, but in a more complex way. Assignment 31 has been written such as the whole program is in the main function while assignment 4 has been divided into small functions. The others suspicious couple seem to share a similar architecture (advised by the teacher) and several system calls, but given that these students were in the same class, we can assume that they kept the piece of code advised by the teacher. This experiment shows that our tool is robust against source code reorganisation.

These experiments were conducted on a 2Ghz Intel Dual Core with 1Gb RAM. The memory footprint never exceeded 40Mb for these two data sets. Note that every pair of documents can be processed in parallel on different CPUs,

	2	1	5	3	12	11	9	10	6	8	4	13	7
2	0.00	0.74	0.99	0.99	0.99	0.99	0.99	1.00	0.99	0.99	1.00	0.99	0.99
1	0.74	0.00	0.98	0.98	0.99	0.99	0.98	0.99	0.97	0.96	0.99	0.97	0.98
5	0.99	0.98	0.00	0.01	0.88	0.90	0.94	0.97	0.96	0.98	0.96	0.95	0.99
3	0.99	0.98	0.01	0.00	0.95	0.95	0.97	0.96	0.93	0.98	0.94	0.95	0.98
12	0.99	0.99	0.88	0.95	0.01	0.86	0.88	0.96	0.93	0.96	0.97	0.93	0.97
11	0.99	0.99	0.90	0.95	0.86	0.00	0.87	0.95	0.95	0.98	0.98	0.95	0.97
9	0.99	0.98	0.94	0.97	0.88	0.87	0.00	0.96	0.95	0.95	0.97	0.97	0.97
10	1.00	0.99	0.97	0.96	0.96	0.95	0.96	0.00	0.93	0.95	0.96	0.97	0.98
6	0.99	0.97	0.96	0.93	0.93	0.95	0.95	0.93	0.00	0.95	0.96	0.98	0.97
8	0.99	0.96	0.98	0.98	0.96	0.98	0.95	0.95	0.95	0.00	0.88	0.88	0.94
4	1.00	0.99	0.96	0.94	0.97	0.98	0.97	0.96	0.96	0.88	0.00	0.94	0.96
13	0.99	0.97	0.95	0.95	0.93	0.95	0.97	0.97	0.98	0.88	0.94	0.00	0.98
7	0.99	0.98	0.99	0.98	0.97	0.97	0.97	0.98	0.97	0.94	0.96	0.98	0.00

Figure 6. Final document distance colored matrix for Haskell source code produced by CS students, using segmentation by newlines, Levenshtein segment distance, 5x5 identity matrix with 0.7 threshold for post-filtering.

	5	2	7	1	13	15	3	8	14	11	10	6	12	9	4
5	0.00	0.00	0.62	0.62	0.62	0.90	0.95	0.90	0.76	0.88	0.92	0.86	0.94	0.96	0.95
2	0.00	0.00	0.62	0.62	0.62	0.90	0.95	0.90	0.76	0.88	0.92	0.86	0.94	0.96	0.95
7	0.62	0.62	0.01	0.01	0.01	0.87	0.87	0.91	0.69	0.84	0.85	0.95	0.96	0.98	0.97
1	0.62	0.62	0.01	0.01	0.01	0.87	0.87	0.91	0.69	0.84	0.85	0.95	0.96	0.98	0.97
13	0.62	0.62	0.01	0.01	0.01	0.87	0.87	0.91	0.69	0.84	0.85	0.95	0.96	0.98	0.97
15	0.90	0.90	0.87	0.87	0.87	0.01	0.60	0.61	0.66	0.81	0.74	0.86	0.94	0.97	0.97
3	0.95	0.95	0.87	0.87	0.87	0.60	0.00	0.75	0.69	0.83	0.82	0.92	0.91	0.96	0.96
8	0.90	0.90	0.91	0.91	0.91	0.61	0.75	0.01	0.62	0.86	0.82	0.86	0.95	0.98	0.96
14	0.76	0.76	0.69	0.69	0.69	0.66	0.69	0.62	0.02	0.62	0.64	0.82	0.95	0.96	0.95
11	0.88	0.88	0.84	0.84	0.84	0.81	0.83	0.86	0.62	0.01	0.81	0.96	0.95	0.96	0.96
10	0.92	0.92	0.85	0.85	0.85	0.74	0.82	0.82	0.64	0.81	0.01	0.92	0.95	0.97	0.97
6	0.86	0.86	0.95	0.95	0.95	0.86	0.92	0.86	0.82	0.96	0.92	0.00	0.95	0.96	0.96
12	0.94	0.94	0.96	0.96	0.96	0.94	0.91	0.95	0.95	0.95	0.95	0.95	0.04	0.92	0.95
9	0.96	0.96	0.98	0.98	0.98	0.97	0.96	0.98	0.96	0.96	0.97	0.96	0.92	0.01	0.96
4	0.95	0.95	0.97	0.97	0.97	0.97	0.96	0.96	0.95	0.96	0.97	0.96	0.95	0.96	0.00

Figure 7. Final document distance colored matrix for Python source code produced by CS students, using the same settings as in Figure 6.

Corpus name	*# Documents*	*# Couples*	*# Suspects*	*# Plagiarised*	**Recall**	**Precision**	F_2 **measure**
HASKELL	13	78	3	3	1.0	1.0	1.0
PYTHON	15	105	20	4	1.0	0.2	0.55
C	19	171	7	4	1.0	0.57	0.87

Table I
EVALUATION ON EACH CORPUS

enabling scalability to larger corpora.

Table V summarizes the evaluation of our tool on the corpus presented above. We consider that we detect plagiarism when the distance between a pair of documents is less than the mean distances of the matrix. Note that, in all evaluations the recall is always maximised which means that we detected all plagiarised documents. Nevertheless, the precision is far from prefect, especially with the Python corpus. The false positive detection is mainly due to the nature of the assignment : some of the students used a source code skeleton given by their teacher and the other ones wrote the entire project from scratch. Our tool detected the students who used the skeleton, but note that the plagiarists were highlighted in figure 7.

VI. DISCUSSION

In the previous section, our tool was tested against many different programming languages. The main difficulty for us is to correctly annotate each corpus manually. It is easily understandable that students don't claim that they resorted to plagiarism.

	9	21	28	23	1	4	31	12	5	25	13	22
9	0.00	0.77	0.80	0.89	0.87	0.80	0.90	0.87	0.81	0.86	0.81	0.84
21	0.77	0.00	0.80	0.92	0.89	0.81	0.84	0.86	0.81	0.80	0.82	0.89
28	0.80	0.80	0.00	0.92	0.92	0.88	0.92	0.88	0.89	0.91	0.92	0.94
23	0.89	0.92	0.92	0.00	0.78	0.80	0.88	0.85	0.88	0.81	0.90	0.85
1	0.87	0.89	0.92	0.78	0.00	0.81	0.87	0.88	0.89	0.88	0.88	0.90
4	0.80	0.81	0.88	0.80	0.81	0.00	0.55	0.69	0.76	0.80	0.88	0.84
31	0.90	0.84	0.92	0.88	0.87	0.55	0.00	0.79	0.83	0.88	0.91	0.91
12	0.87	0.86	0.88	0.85	0.88	0.69	0.79	0.00	0.85	0.87	0.91	0.85
5	0.81	0.81	0.89	0.88	0.89	0.76	0.83	0.85	0.00	0.81	0.89	0.92
25	0.86	0.80	0.91	0.81	0.88	0.80	0.88	0.87	0.81	0.00	0.89	0.90
13	0.81	0.82	0.92	0.90	0.88	0.88	0.91	0.91	0.89	0.89	0.00	0.90
22	0.84	0.89	0.94	0.85	0.90	0.84	0.91	0.85	0.92	0.90	0.90	0.00

Figure 8. Final document distance colored matrix for C source code produced by CS students, using the same settings as in Figure 6.

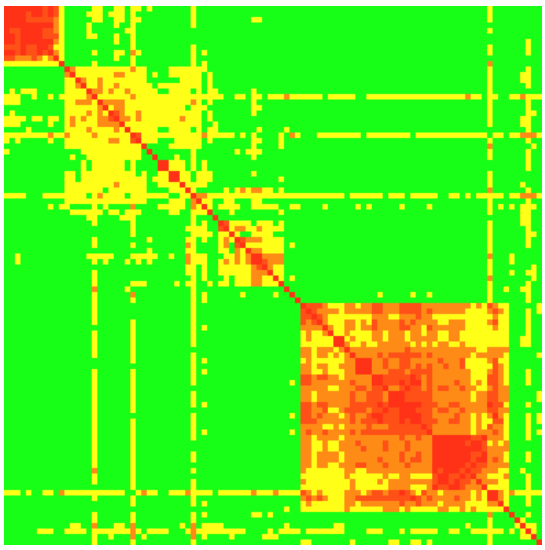


Figure 9. Final document distance colored matrix for a C++ source code produced by CS students

Figure VI presents the results of plagiarism detection on 98 files written in C++. Each pixel of the image represents a cell on the above figures. Annotating such a corpus means comparing each document against each other. Even if we spend only 10 minutes for each comparison, the corpus annotation would take more than 13 hours. This illustrates the complexity of plagiarism detection well and the need for a plagiarism detection tool. Interestingly, in this corpus many couples were plagiarised but none of the teachers could manually detect any of the fraud.

Corpus annotation is a time consuming task and as far as we know there are no source code plagiarism detection challenges or annotated assignments freely available. For all these reasons, we couldn't formally validate our tool against much more assignments.

We think it would be a great initiative to organise a source code plagiarism detection challenge. This way different approaches from different communities (source code analysis, multilingual natural language plagiarism, data-mining, etc.) could be compared with the ultimate goal to improve the interactions between these communities.

VII. Conclusion and future work

We presented a source code plagiarism detection framework which allowed us to discover previously undetected frauds in some corpora, even when the students were given a common code base to start with.

This work is still in progress. We have to work on an interactive corpus summary to ease the corrector's work, allowing him to explore the corpus at both document and segment levels, and to view the results using different pipelines.

We interfaced the framework with the homework repository of the University of Caen's computer science department. This way, every teacher can now use the framework on his own. Furthermore, the number of available assignments for our experiments increases and we hope to be able to build annotated source code corpora in order to set up a plagiarism detection challenge.

The experiments presented in this paper show promising results for a specific pipeline configuration. In order to fairly compare different pipelines, we built a modular implementation of the framework. We aim to benchmark different pipeline configurations in order to find the best ones and to improve the results presented in this paper (mainly the precision). The modular implementation of the framework is publicly accessible online [2].

References

[1] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *ESEC/SIGSOFT FSE 2005: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT*

[2] http://code.google.com/p/pypometre

International Symposium on Foundations of Software Engineering, 2005, pp. 187–196.

[2] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSM 1998: Proceedings of the 12th International Conference on Software Maintenance*, 1998, pp. 368–377.

[3] J. Krinke, "Identifying similar code with program dependence graphs," in *WCRE 2001: Proceedings of the 8th Working Conference on Reverse Engineering*, 2001, pp. 301–309.

[4] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *WCRE 1995: Proceedings of the 2nd Working Conference on Reverse Engineering*, 1995, pp. 86–95.

[5] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[6] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching," *Journal of Software Maintenance*, vol. 18, no. 1, pp. 37–58, 2006.

[7] R. Wettel and R. Marinescu, "Archeology of code duplication: Recovering duplication chains from small duplication fragments," in *SYNASC 2005: Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2005, pp. 63–70.

[8] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Software Eng.*, vol. 33, no. 9, pp. 577–591, 2007.

[9] G. Whale, "Identification of program similarity in large populations," *The Computer Journal*, vol. 33, no. 2, pp. 140–146, 1990.

[10] M. Wise, "YAP3: Improved detection of similarities in computer program and other texts," *Twenty-Seventh SIGCSE Technical Symposium*, pp. 130–134, 1996.

[11] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with jplag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.

[12] G. Malpohl, "Jplag, Detecting Software Plagiarism," http://www.ipd.uni-karlsruhe.de/jplag/.

[13] S. Schleimer, D. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD*. ACM New York, NY, USA, 2003, pp. 76–85.

[14] M. Freire and M. Cebrian, "Design of the ac academic plagiarism detection system," Technical report, Tech. rep., Escuela Politecnica Superior, Universidad Autonoma de Madrid, Madrid, Spain, Tech. Rep., 2008.

[15] G. Cosma, "An approach o source-code plagiarism detection and investigation using latent semantic analysis," Ph.D. dissertation, University of Warwick, 2008.

[16] J. W. Son, S.-B. Park, and S.-Y. Park, "Program plagiarism detection using parse tree kernels," in *PRICAI 2006: Proceedings of the 9th,Pacific Rim International Conference on Artificial Intelligence*, 2006, pp. 1000–1004.

[17] N. Göde, "Evolution of type-1 clones," in *SCAM 2009: Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009, pp. 77–86.

[18] R. Tiarks, R. Koschke, and R. Falke, "An assessment of type-3 clones as detected by state-of-the-art tools," in *SCAM 2009: Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009, pp. 67–76.

[19] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in *ICSE 2007: Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 106–115.

[20] T. Urvoy, T. Lavergne, and P. Filoche, "Tracking web spam with hidden style similarity," *AIRWeb 2006 Program*, p. 25, 2006.

[21] Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, pp. 707–710, 1966.

[22] R. Cilibrasi and P. Vitanyi, "Clustering by compression," *IEEE Transactions on Information theory*, vol. 51, no. 4, pp. 1523–1545, 2005.

[23] H. Kuhn, "The hungarian method for the assignment problem," *Naval Res. Logist. Quart.*, vol. 2, pp. 83–97, 1955.

[24] J. Véronis, "From the Rosetta stone to the information society," *Parallel Text Processing-Alignment and Use of Translation Corpora*, pp. 1–24, 2000.