Comparison of Clang Abstract Syntax Trees using string kernels

Conference or Workshop Item

Accepted Version

Comparing Clang Abstract Syntax Trees using String Kernels

Raul Torres, Julian M. Kunkel, Thomas Ludwig
Department of Informatics,
Universität Hamburg,
20146–Hamburg, Germany
Email: raul.torres@informatik.uni-hamburg.de,
juliankunkel@googlemail.com,
ludwig@dkrz.de

Manuel F. Dolz
Department of Computer Science,
Universidad Carlos III de Madrid,
28911–Leganés, Madrid, Spain
Email: mdolz@inf.uc3m.es

***Abstract*—Abstract Syntax Trees (ASTs) are intermediate representations widely used by compiler frameworks. One of their strengths is that they can be used to determine the similarity among a collection of programs. In this paper we propose a novel comparison method that converts ASTs into weighted strings in order to get similarity matrices and quantify the level of correlation among codes. To evaluate the approach, we leveraged the corresponding strings derived from the Clang ASTs of a set of 100 source code examples written in C. Our kernel and two other string kernels from the literature were used to obtain similarity matrices among those examples. Next, we used Hierarchical Clustering to visualize the results. Our solution was able to identify different clusters conformed by examples that shared similar semantics. We demonstrated that the proposed strategy can be promisingly applied to similarity problems involving trees or strings.**

I. Introduction

Computer programs exhibit similarities that can be detected before, upon and after execution time. Being similar means sharing syntactical or semantical structure in a significant proportion. Programs that are similar tend to behave in similar manner too. This can be utilized, for example, in the analysis and improvement of the overall performance of a set of programs by focusing on finding patterns that behave similar but have a different performance. The detection of program similarities has been identified as an emerging topic in software engineering areas [1].

From the perspective of code sharing, finding similar code can, for example, assist the programmer in finding code that is already implemented in a library and hinting users to utilize the library instead of recoding. Developers might be even able to find syntactically dissimilar, yet more efficient, versions of their algorithms with similar semantics. The optimizations implemented in an algorithm might benefit similar programs, once the similarity relation has been established.

Furthermore, the search could be specialized on finding common mistakes at designing or writing programs, the so-called *code smells* [2]. A possible way to do this could be by comparing our own code against a collection of code excerpts already recognized as containers of code smells, and subsequently applying machine learning to extract knowledge from the similarity scores.

On the counter direction of code sharing, we have code plagiarism, a phenomena that has increasingly motivated research in program similarity. Starting from Computer Science lectures where teachers need to verify the originality of their students' work, till the protection of copyrighted code that might have been illegally used, plagiarism detection is a very active research field.

Abstract Syntax Trees (ASTs) are one of the most commonly used Intermediate Representations (IRs) inside compiler infrastructures (e.g. LLVM's Clang AST). They act as the central data structure where almost most of the transformations and analysis are done. Though ASTs can be used to determine how similar two code fragments are, there is not so much available work related to the use of string kernels to compare data structures of this particular nature.

This work extends the state-of-the-art in detecting code similarities using string kernels. In this sense, the contributions in this paper are twofold: a) we propose a strategy to convert ASTs from Clang into flat weighted strings; this way we can use string kernels for the comparison process; b) we propose a novel string kernel function called *kast1 spectrum kernel*; this kernel delivers a similarity score among two strings that is determined by the contributions of matching substrings.

This paper is organized as follows: in Section II, the basic foundations of Compiler's Intermediate Representations and String Kernels are presented. Section III revisits some related works in the area. Section IV explains the rational behind both the proposed string representation and the kernel function. The evaluation of the approach is conducted in Section V. Finally, Section VI summarizes the results and details possible future paths for the current research efforts.

II. Background

In this section we describe the main topics related to this research: compiler intermediate representations, strategies and techniques for code similarity, background on string.

A. Compiler Intermediate Representations

According to Torczon and Cooper [3], the Intermediate Representation (IR) of a program is the central data structure in

a compiler. Around it, analysis, transformations and optimizations are performed. Complex compiler infrastructures might work with different interconnected IRs, some of them closer to the source code, others closer to the machine instruction level. IRs can be classified in the following three broad categories:

- *Graphical IRs* store the program information in a graph-like data structure.
- *Linear IRs* are simple linear sequences of operations, similar to machine code.
- *Hybrid IRs* combine elements of the previous two categories.

Among the graphical IRs, Abstract Syntax Trees (ASTs) are widely used. ASTs are defined as contractions of parse trees where most non-terminal symbols are ignored while the precedence and the meaning of the expressions are preserved, thus saving space. Their level of abstraction is not far from the original source code.

In this sense, we highlight the LLVM [4] compiler infrastructure, a framework that uses different IRs to perform program analysis, transformation and code generation. Among the variety of tools available under this infrastructure, there exists Clang [5], a frontend for C/C++/Objective C programs. Upon compilation, Clang first captures the syntactical structure of the program in an AST [5]. Afterwards, the AST is traversed to generate the linear IR that is used by LLVM to perform transformation and optimizations, and finally generate machine specific code. There are three core classes of AST nodes: Declarations, Statements, and Types [6]; all the class hierarchy of Clang inherit directly or indirectly from them. For example, for the expression: $a = b \times c + d \div e$, the abbreviated Clang AST would look similar to the diagram in Fig. 1.

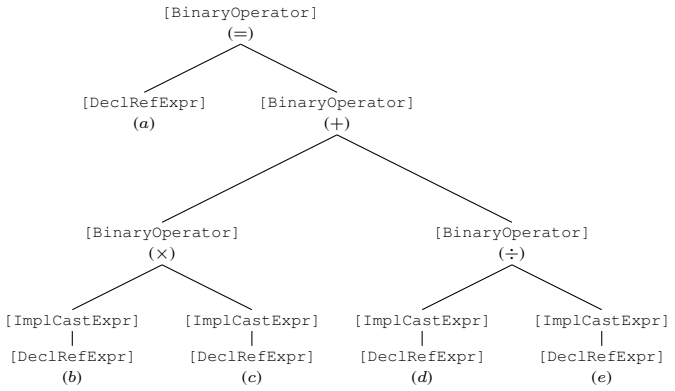


Fig. 1. Example of abbreviated Clang AST for the expression $a = b \times c + d \div e$.

B. Code similarity

Code plagiarism is an emerging topic that increasingly motivated research in code similarity. For instance, the work of Beth [7] listed the general strategies used to obscure plagiarism: comment alteration, whitespace padding, identifier renaming, code reordering and algebraic expressions. In general, code clones are a measure of plagiarism. In this sense, Dang et al. [8] defined code clones as portions of code with high similarity in syntax or semantics. They listed four different types of code clones:

- *Type-1:* this type of clones stand for pieces of code containing differences in the layout, spaces and comments.
- *Type-2:* these codes also present differences in data types and identifiers.
- *Type-3:* this clone types also include additions, modifications and deletions of lines of code.
- *Type-4:* represent codes that present different implementation but the same functionality.

On the other hand, the work of Vislavski et al. [9] made a classification of level of approaches to detect code clones. These approaches are the following:

- *Textual approach:* this approach treats code merely as text and it is useful for Type-1 clones.
- *Lexical approach:* at this approach level, tokens are the unit of comparison and add a bit more of sophistication to the textual approaches.
- *Syntactical approach:* this approach is characterized by the usage of either ASTs or Source Code Metrics.
- *Semantic approach:* this approach is different to the previous ones, as they make use of more complex tools e.g. Control and Data Flow Analysis.

Following this classification, the higher the level, the better the detection can be made. In this paper, we leverage a syntactical approach based on the AST associated to a given program.

C. String kernels

In machine learning and data mining, a string kernel is a kernel function that operates on strings, i.e. finite sequences of symbols that need not be of the same length. String kernels can be intuitively understood as functions measuring the similarity of pairs of strings: the more similar two strings a and b are, the higher the value of a string kernel $K(a, b)$ will be. In particular, string kernels check the number of shared substrings among a collection of strings [10]. These substrings should comply with certain weighting factors, which produce different kernel functions:

- The *bag-of-characters* kernel only takes into account single-character matching.
- The *bag-of-words kernel* searches for shared words among strings.
- The *k-spectrum kernel* only counts sub-strings of length k [11].
- The *blended spectrum kernel* only counts sub-strings whose length are less or equal to a given number k [12].

In this work, we leverage the aforementioned string kernels in the domain of code plagiarism in order to detect similarities between string-represented ASTs from a collection of programs.

III. Related Work

Given the aforementioned background, the string kernel version proposed in this paper extends the current state-of-the-art in the field of code similarity detection. This is based on a previous kernel proposed by Torres et al. [13], where the

authors proposed a string kernel and a string representation for the detection of patterns in I/O traces.

In a similar way, in the work of Fu et al.[14] it was proposed a weighted kernel method for source code plagiarism detection based on ASTs. The weights of each AST node were determined by a technique called TF-IDF (Term Frequency-Inverse Document Frequency). The authors claimed that their method resulted on improved detection in comparison with two other plagiarism detection tools, namely JPlag [15] and Sim [16].

A tree kernel-based approach for clone detection can be found in [17]. Similar to their work, we used in our research an abstraction for `while` and `for` control flow instructions. In their work, the similarity score of two nodes was limited to six values that were adjusted by the authors, and depended on the proximity in terms of the class/context system they defined. Experimental evaluation on a set of Java code examples resulted on proper detection of *Type-1*, *Type-2* and *Type-3* clones.

Research in this direction, e.g. Bandara et al. [18], used an unsupervised technique called *sparse auto-encoder* to extract the features from a piece of code. Logistic regression was used to predict the author of a code segment. In the direction of code smells detection, Danphitsanuphan et al. [19] proposed an approach to find a relationship between code smells and software structure bugs.

On the other hand, the work of Park at al. [20] used a combined approach of parse trees and function-call graphs. They created a composite kernel, with the parse tree kernel and the graph kernel. Similar works, such as by Sharma et al. [21], used a bag-of-words approach to characterize intrusion, where the words corresponded to the system calls that the program performed. In the work of Wang et. al [22], a new methodology was developed to detect Platform-Specific Code Smells (PSCSs) in High Performance Computing applications using Abstract Syntax Tree (AST) and XML.

IV. Methodology

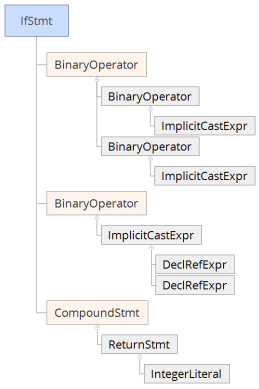
In this section, we describe our method for converting Clang ASTs into weighted strings. We also present a novel kernel function to compare the resulting strings.

A. Creation of strings from Abstract Syntax Trees

1) From trees to strings: To create the strings, ASTs are traversed in a pre-order fashion. During the traversal, the identifiers of the visited nodes are appended as tokens in the string, except from comment nodes and their eventual children. Basically, these tokens consists of a literal part and a weight value. The literal part is the class name of the node given by Clang AST, while the weight is the number of consecutive occurrences of the token, which at first always equals to 1.

In order to achieve generality, we transform loop statement classes, such as `ForStmt` and `WhileStmt` into the new token `[LoopedStmt]`. Additionally, to preserve information about the tree structure, we introduce a new token that does not correspond to any node of the AST but gives a notion of distance between nodes; this node is the `[LEVEL_UP]` token and represents the change to an upper level when doing the pre-order traversal. Its weight is simply the amount of levels jumped until the next new node is found. Notice that there is no need for a token to indicate a change to a lower level. This is due to the fact that in the pre-order traversal the number of levels jumped from a parent to a child is always 1. This effect is implicitly expressed when two tokens are written one after the other.

Fig. 2 shows a worked example of conversion from an AST to a string; first the AST (Fig. 2a) is traversed in a pre-order fashion; during the traversal its tokens are extracted (Fig.2b) and appended at the end of the final string (Fig. 2c).



(a) Abstract Syntax Tree.

Tokens	Repetitions
[IfStmt]	1
[BinaryOperator]	1
[BinaryOperator]	1
[ImplicitCastExpr]	1
[LEVEL_UP]	2
[BinaryOperator]	1
[ImplicitCastExpr]	1
[LEVEL_UP]	3
[BinaryOperator]	1
[ImplicitCastExpr]	1
[DeclRefExpr]	1
[LEVEL_UP]	1
[DeclRefExpr]	1
[LEVEL_UP]	3
[CompoundStmt]	1
[ReturnStmt]	1
[IntegerLiteral]	1
[LEVEL_UP]	4

(b) Extracted tokens.

$$[\texttt{IfStmt}]_1[\texttt{BinaryOp}]_1[\texttt{BinaryOp}]_1 \ldots [\texttt{LEVEL\_UP}]_4$$

(c) Final string.

Fig. 2. Creation of a string of tokens from an Abstract Syntax Tree.

2) Compression of strings: Once the string is generated, space can be saved when a set of consecutive tokens follows a pattern that can be expressed as a single token. The resulting token will have, as weight, the summation of weights of all involved tokens, in order to preserve the original weight of the string. This space-saving technique is applied in order to compress the string when it is traversed from left to right. The following transformations are applied depending on the form of the string:

1) Consecutive tokens with the same literal part are represented as a single token and their weights are summed up. For instance:

$$[\texttt{BinaryOp}]_1[\texttt{BinaryOp}]_1[\texttt{BinaryOp}]_1[\texttt{BinaryOp}]_1 \downarrow \textbf{[BinaryOperator]}_4$$

Rationale: The motivation of this transformation is merely to save space.

2) Tokens representing cast expressions, parent expressions and function calls are deleted, but their weights are added up to the weight of the next subsequent token. For example:

$$[\texttt{CStyleCastExpr}]_1[\texttt{CallExpr}]_1[\texttt{ImplicitCastExpr}]_1 \textbf{[DeclRefExpr]}_1 \downarrow \textbf{[DeclRefExpr]}_4$$

Rationale: Cast expressions are used to assist the compiler in the conversion of values between similar data types; although this is important at compilation time, it was proved to be cumbersome in the similarity study. This is because the process introduced different tokens in two strings that otherwise would be semantically identical, obstructing the achievement of a suitable abstraction level, necessary to establish a similarity measure among code examples. Likewise, parent expressions are internal constructions of the Clang AST, whose omission improves abstraction.
The same is the case for function calls; keeping a token for a function call also broke the structure of string segment. The idea behind these omissions was to obtain the largest possible matching substrings. Experimentation showed that ignoring these tokens helped to increase the identification of *Type-2* clones.

3) All tokens between a Declaration Statement token (`[DeclStmt]`) and a `[LEVEL_UP]` token are deleted but their weights are summed up to the weight of the former. For instance:

$$\texttt{[DeclStmt]}_1\,\texttt{[VarDecl]}_1\,\texttt{[DeclRefExpr]}_1\,\texttt{[LEVEL\_UP]}_4 \downarrow \texttt{[DeclStmt]}_3\,\texttt{[LEVEL\_UP]}_4$$

Rationale: Similar to cast expressions, the tokens of a declaration statement introduced a level of detail that made the abstraction difficult. Omitting them eased the detection of similar declaration blocks.

4) If two pairs of tokens have the same literal part, they are collapsed as one pair and their weights are added up to the corresponding token. For example:

$$\begin{matrix}\texttt{[IntegerLiteral]}_1\texttt{[LEVEL\_UP]}_5\\ \texttt{[IntegerLiteral]}_1\texttt{[LEVEL\_UP]}_2\end{matrix} \downarrow \texttt{[IntegerLiteral]}_2\texttt{[LEVEL\_UP]}_7$$

Rationale: The motivation in this case is also space saving, as the resultant string is semantically equivalent after this transformation.

B. String comparison

This kernel is based on the *kast spectrum kernel*, presented by Torres et al. [13]. Given two weighted strings A and B, the kernel proposed in this paper must follow the conditions given below:

1) The algorithm precises a minimum weight value as parameter (from here on referred simply as **cut weight**).
2) The aim is to find the longest matching substrings of A and B, whose weights are greater than or equal to the cut weight. They are called *valid matching substrings*. Invalid matching substrings have a weight value that is smaller than the cut weight, and are hence ignored.
3) A *valid matching substring* can appear more than once in each string.
4) A *valid matching substring* must not be a substring of another *valid matching substring* in at least one of the original strings.

In order to find the longest *valid matching substrings* efficiently, the algorithm starts searching for matches of maximum size. The maximum size is the number of tokens of the shortest string of the comparison (A or B). This size is reduced progressively until 1, but always checking that the substrings weights are equal or above the cut weight. A copy of each one of the original strings is used to mark down the already found *valid matching substrings*. Potential *valid matching substrings* have to be checked against those copies to assure that condition 4 is met.

1) The cut weight parameter: One the one hand, the cut weight selection has an effect on the computation cost, as the algorithm takes into account all the substrings whose weights are equal or greater than the cut weight. If the cut weight is 1, all substrings have to be compared. An increase on the cut weight allows the filtering of substrings with small weights. If the cut weight is closer to the weight of the strings (A or B), the amount of substrings having a valid weight is reduced considerably. Hence, the higher the cut weight, the cheaper the computation.

On the other hand, the cut weight controls the size of the consecutive parts that are shared. If the cut weight is equal to the smallest weight between A and B, we are only accepting that either A or B is contained fully on the other string. This might be useful to perform code search: specific segments of code can be survey in library or a project. A decrease on the cut weight permits that segments of the strings can be considered as *valid matching substrings*. If the cut weight is closer to 1, short sequences of tokens will contribute on the similarity score. In this sense, a trade-off has to be found. Experimentation showed that cut weight values up to 32 were optimal for the construction of good similarity matrices.

The kernel presented in this paper is an asymmetric kernel, which means that the kernel value depends on the order in which two string are compared. Because the substrings conforming the first string are always the base for the search, there might be a difference when the strings are swapped. Given the kernel function starts always searching for the longest matching substrings, these changes happen with smaller matching substrings only, so the the difference between the kernel values is not significantly high. This also shows that the smaller is the cut weight, the highest is the probability of this difference to appear.

2) The Kast1 Spectrum Kernel: Consider the strings A and B of Fig. 3, and a cut weight value of 4. The first *valid matching substring* with the longest size (S_1) is found once in A and twice in B (see Fig. 3). The second longest *valid matching substring* (S_2) is found twice in A and twice in B (see Fig.4). This substring appears at least once as an independent substring in one of the strings, hence complying with condition 4. Notice that an extra occurrence is ignored because its weight is smaller than 4. The last and shortest *valid matching substring* (S_3) is found twice in A and twice in B (see Fig. 5). As the substring appears as an independent case in both strings, it complies with condition 4 as well. Here also an extra occurrence is ignored due to a smaller weight.

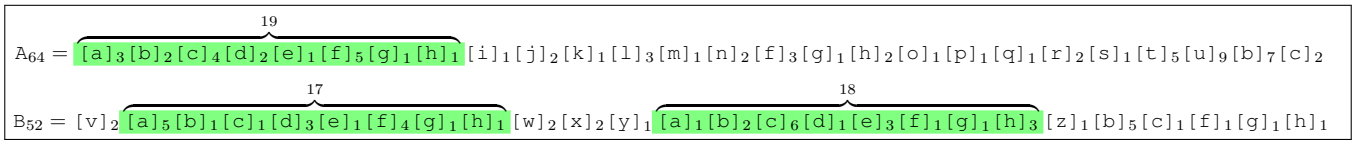


Fig. 3. S_1 is the largest substring found on both examples.

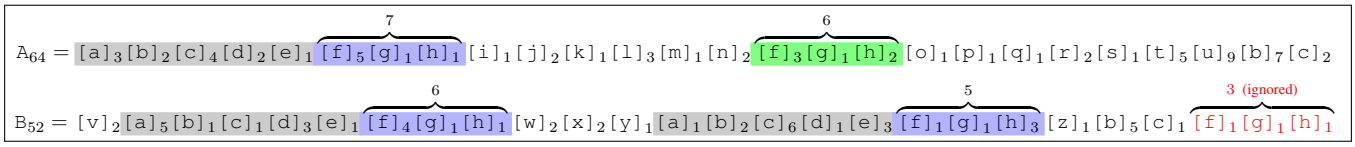


Fig. 4. S_2 appears once as an independent case.

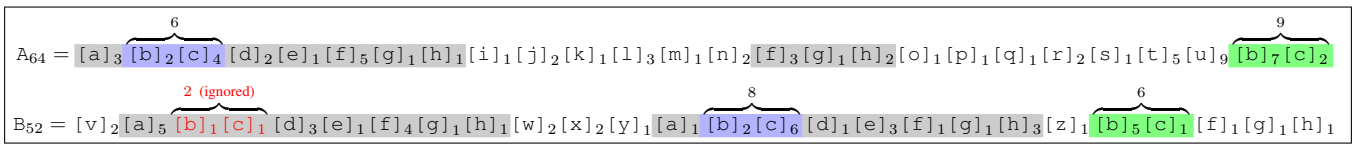


Fig. 5. S_3 appears twice as an independent case.

The *kast1 spectrum kernel* has the following definition:

- Each *valid matching substring* embeds a new feature for A and B. Hence, the size of the new embedding vector for both strings is equal to the number of *valid matching substrings*.
- Only the weight of the independent *valid matching substrings* is taken into account to build the feature value, which corresponds to the summation of these weights.
- If the string does not present and independent occurrence of a particular *valid matching substring*, the feature value is set to 1, to avoid zero values when calculating the inner product.
- The kernel value corresponds to the inner product of the new feature vectors of A and B.

Example:

Let A and B be the same strings of the previous example (see Fig. 3). In this case the function of *the kast1 spectrum kernel*, $weight\_k1_{w\geq n}(S)_A$ returns, either:

- the summation of the weights of all the independent matching instances of S in A whose weight is greater than or equal to n,
- or 1 if there are no independent substrings.

This results in three cases of matching substrings (see Fig. 3, 4 and 5).

For instance, for a cut weight of 4 ($n = 4$), the respective weights of each feature in A are calculated with:

$$weight\_k1_{w\geq 4}(S_1)_A = 19 \tag{1}$$

$$weight\_k1_{w\geq 4}(S_2)_A = 6 \tag{2}$$

$$weight\_k1_{w\geq 4}(S_3)_A = 9 \tag{3}$$

Therefore, the new embedding feature vector for A is:

$$f1_{w\geq 4}(A) = \{19, 6, 9\} \tag{4}$$

Notice in Fig. 4 that S_2 does not appear as an independent *valid matching substring* in B. Thus, the partial feature value is set to 1 (see Eq. 6). The respective weights of each feature in B are calculated with:

$$weight\_k1_{w\geq 4}(S_1)_B = 17 + 18 = 35 \tag{5}$$

$$weight\_k1_{w\geq 4}(S_2)_B = 1 \tag{6}$$

$$weight\_k1_{w\geq 4}(S_3)_B = 6 \tag{7}$$

Thus, the new embedding feature vector for B is:

$$f1_{w\geq 4}(B) = \{35, 1, 6\} \tag{8}$$

The function $k1_{w\geq n}(A, B)$ returns the evaluation of the kernel value between A and B; this is no more than the inner product of the new feature vectors:

$$k1_{w\geq 4}(A, B) = \langle f1_{w\geq 4}(A), f1_{w\geq 4}(B)\rangle = 725 \tag{9}$$

The function $\bar{k1}_{w\geq n}(A, B)$ is the normalized version of the kernel. A normalization step will use the weights of each string:

$$\begin{aligned}\bar{k1}_{w\geq 4}(A, B) &= \frac{k1_{w\geq 4}(A, B)}{\sqrt{k1_{w\geq 4}(A, A) \times k1_{w\geq 4}(B, B)}} \\ &= \frac{k1_{w\geq 4}(A, B)}{weight\_k1_{w\geq 4}(A) \times weight\_k1_{w\geq 4}(B)}\end{aligned} \tag{10}$$

$$\bar{k1}_{w\geq 4}(A, B) = \frac{725}{64 \times 52} = \frac{725}{3328} \approx 0.2178 \tag{11}$$

Therefore, it is possible to say that these two strings are 21.78% similar. Thanks to this kernel, we are able to measure the similarity of two arbitrary strings. Assuming that these strings are conversions of the AST of two given programs, we are therefore able to obtain a similarity index between both programs.

V. Experimental Evaluation

The experimental evaluation was designed to assess the capabilities of the proposed kernel to recognize classes of functions and how the clone types were organized inside each class. It was not the focus of this research the design of a code plagiarism tool, that was the reason why it was not evaluated against the plethora of code detection tools available out there. The main focus was to provide experimental proof that this new kernel accomplishes the goals in a similar or even better way than other kernels from the literature.

Thus, in this section we tested the proposed kernel function on a set of code examples and compared the results against two kernel functions from the state-of-the-art: the *blended spectrum kernel* [12] and the *kast spectrum kernel* [13].

A. Experiment configuration

For these experiments, we used 20 different functions. These functions were divided in four broad classes:

- *Class A (String Matching Functions):* They are string kernels of the literature, but the name was changed to avoid confusion when referring to the kernels used in this study.
  - K-spectrum: matching substrings of size k.
  - Blended spectrum: matching substrings of size k or less.
  - Bag-of-characters: matching substrings of size 1.
  - Bag-of-words: delimiter based.
  - Bag-of-sentences: similar to the latter but with two delimiters, one for the opening, one for the closing.
- *Class B (Sort Functions):*
  - Bubble sort.
  - Insert sort.
  - Selection sort.
  - Heap sort.
  - Merge sort.
- *Class C (3D Stencils):* Stencils are operations performed on a structured grid, where the value of a cell is calculated using the values of the surrounding cells. In this experiment, the initial value of the cell itself was always taken into account. The operation was the summation of all values.
  - Compact stencil: compact stencils take into account only the values of the neighboring cells.
  - Side stencil: this one only takes into account neighboring cells sharing the same position in two axes.
  - Edge stencil: this stencil only takes into account neighboring cells sharing the same position in only one axis.
  - Vertex stencil: it only takes into account neighboring cells on a diagonal position.
  - Non-compact stencil 1 layer: Non-compact stencils go a few layers further the neighboring cells.
- *Class D (2D Stencils):* Similar to the previous class, but with the number of dimensions reduced by one. In this case there is no room for a Side stencil.
  - Compact stencil.
  - Edge stencil.
  - Vertex stencil.
  - Non-compact stencil 1 layer.
  - Non-compact stencil 2 layers.

It was also the interest of this paper to study the structure of clone types inside each function class; for that reason, each function was implemented in five different variants with the same functionality: original version, *Type-1*, *Type-2*, *Type-3* and *Type-4* clones. This resulted then in a set of 100 examples for the study, whose size ranged from 32 to 124 lines of code.[1]

We leveraged the Clang AST of each code sample and converted it into a string. We tested the following cut weight range: $\{2^0, 2^2, ..., 2^n\}$ for $n = 9$. Note that when the computed matrices present negative eigenvalues, these eigenvalues were replaced by zeros and the matrices were recalculated using these new eigenvalues. All the similarity matrices were analyzed with Hierarchical Clustering using the simple linkage method.

B. Baseline kernel 1: Blended Spectrum Kernel

Given the particular form of the proposed string representation, where a group of subsequent tokens can encode more meaningful information than a single one, we discarded the *bag-of-characters* and the *bag-of-words* kernels. The experimental evaluation showed also that the *k-spectrum kernel* was not successful at finding an acceptable clustering, a task where the *blended spectrum kernel* had a better performance.

The best results with this kernel were obtained when using a cut weight of 16. Hierarchical Clustering separated the examples in three clusters (see Fig. 6):

- Class A: String kernels.
- Class B: Sort functions.
- Class C and D: 3D and 2D stencils.

There were three examples of Class B, all of them *Type-4* clones, that were clustered within Class A. It is important to notice that classes C and D were detected as single cluster, which reflected the intuitive similarity among both classes. The experiment also showed clearly that the ranking in distances for clone types with respect to the original version, corresponded most of the times to the expectation according to the theory. *Type-1* clones were almost overlapped with the original versions.

C. Baseline kernel 2: Kast Spectrum Kernel

Experimentation showed a significant advantage of the usage of the *kast spectrum kernel* over the first baseline kernel, as no misplaced examples were found. For the *kast spectrum kernel*, the same basic scheme of clusters was achieved by Hierarchical Clustering, but with larger intra-cluster distances and a cut weight of 64 (see Fig. 7). In this case as well, the ranking in distances to the original version corresponded to the expectation according to the definition of clone types.

[1] Available under https://git.wr.informatik.uni-hamburg.de/raul.torres/kast_test_functions.

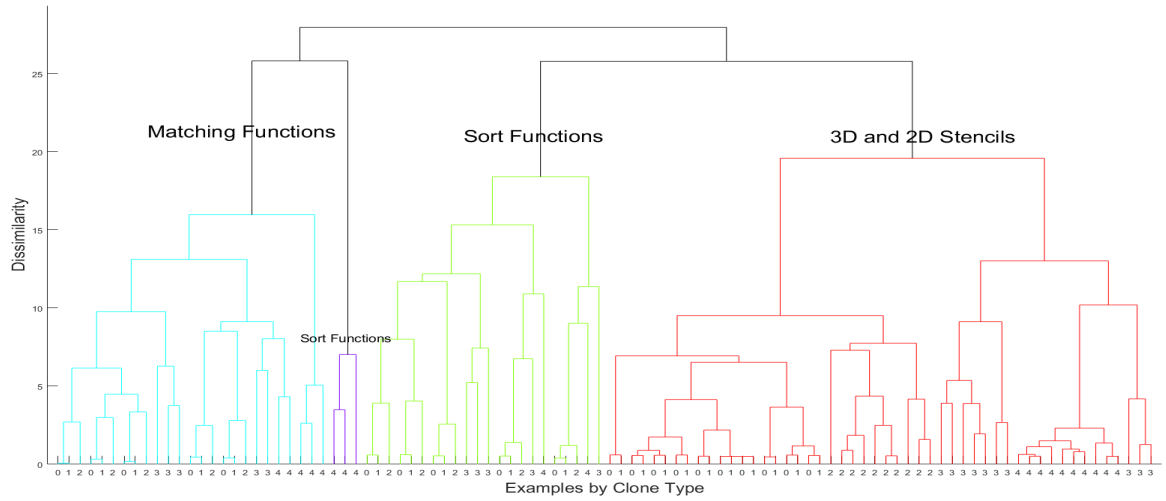


Fig. 6. Hierarchical clustering for Blended Spectrum Kernel using ASTs (cut weight = 16).

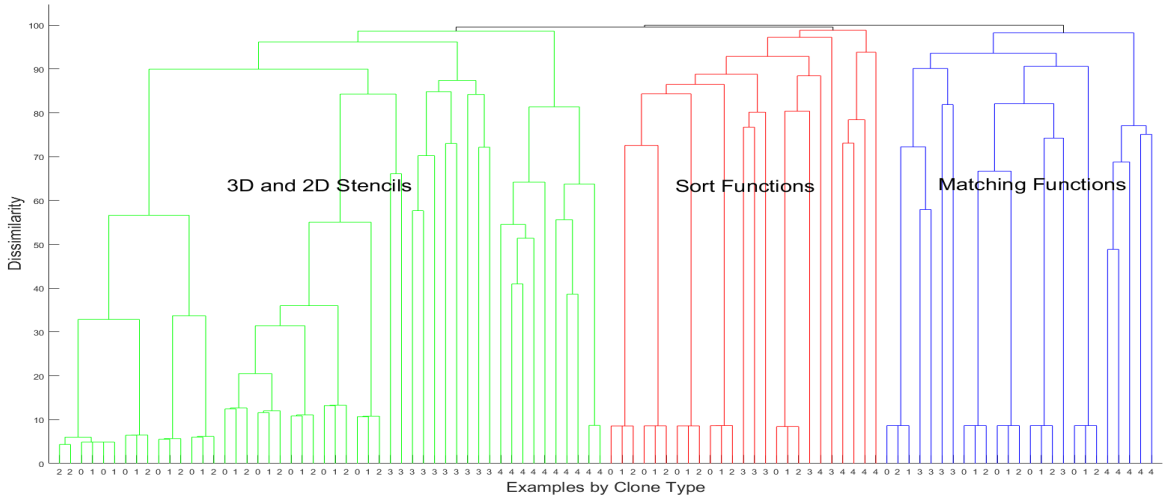


Fig. 7. Hierarchical clustering for Kast Spectrum Kernel using ASTs (cut weight = 64).

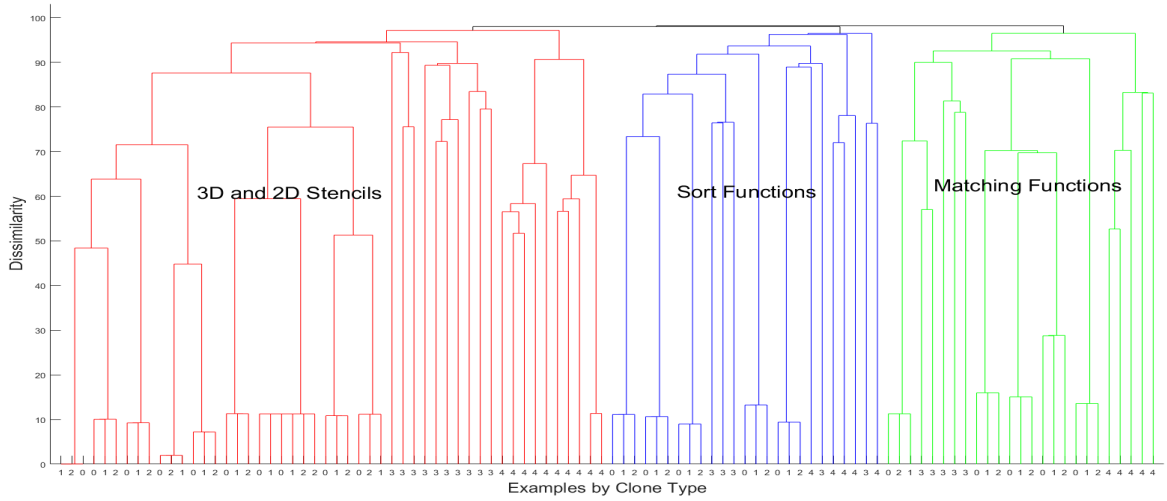


Fig. 8. Hierarchical clustering for Kast1 Spectrum Kernel using ASTs (cut weight = 16).

Type-1 and *Type-2* clones were found almost overlapped to the original version, while *Type-3* and *Type-4* clones were situated at further distances.

D. Our solution: Kast1 Spectrum Kernel

The experiment showed that the kernel here proposed (*kast1 spectrum kernel*) yielded a better cluster separation than the first baseline kernel. The results were similar to those obtained with the *kast spectrum kernel*. The best results for Hierarchical Clustering were obtained using a cut weight of 16, showing no misplaced examples on any of the clusters (see Fig. 8). For the *kast1 spectrum kernel*, the usual basic scheme of clusters was achieved. Another common point was that the ranking in distances for clone types with respect to the original version corresponded to the expectation according to the theory. The intra-cluster distances were larger than those of the first baseline kernel. With this kernel, there were also no misplaced examples and the original version of a function and its corresponding *Type-1* and *Type-2* clones were found almost overlapped.

VI. Conclusions and Future Work

In this paper we showed how to convert ASTs (from Clang) into weighted strings to subsequently compare them. We also proposed a novel kernel function to perform such comparison: the *kast1 spectrum kernel*.

To evaluate our comparison method, we analyzed a set of 100 code examples written in C divided in four broad categories of functions. The proposed *kast1 spectrum kernel* and the *kast spectrum kernel* from the literature had similar clustering performance when using Hierarchical Clustering, yielding better results than the *blended spectrum kernel*. They showed a consistent formation of three clusters: string kernels, sorting functions and stencils (3D and 2D) without misplaced examples. These results indicate that this novel comparison method can be promisingly utilized to find similarities in source code snippets.

Future efforts on this research will focus on the linear intermediate representation delivered by the LLVM Compiler Infrastructure [4].

Acknowledgment

Raul Torres would like to acknowledge the financial support from the Colombian Administrative Department of Science, Technology and Innovation (Colciencias) as well as the mathematical advisory received from Ruslan Krenzler.

References

[1] S. Cesare and Y. Xiang, *Software Similarity and Classification*. Springer-Verlag London, 2012.

[2] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 381–. [Online]. Available: http://dl.acm.org/citation.cfm?id=942800.943571

[3] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[4] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[5] "Clang: A c language family frontend for llvm." [Online]. Available: https://clang.llvm.org/index.html

[6] B. C. Lopes and R. Auler, *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

[7] B. Beth, "A comparison of similarity techniques for detecting source code plagiarism," 2014.

[8] S. Dang and S. A. Wani, "Performance evaluation of clone detection tools," 2015.

[9] T. Vislavski, Z. Budimac, and G. Rakic, "Towards the code clone analysis in heterogeneous software products," in *Proceedings of the Fifth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, Budapest, Hungary, August 29-31, 2016.*, 2016, pp. 89–96. [Online]. Available: http://ceur-ws.org/Vol-1677/paper11.pdf

[10] S. V. N. Vishwanathan and A. J. Smola, "Fast kernels for string and tree matching," in *Proceedings of the 15th International Conference on Neural Information Processing Systems*, ser. NIPS'02. MIT Press, 2002, pp. 585–592. [Online]. Available: http://dl.acm.org/citation.cfm?id=2968618.2968691

[11] C. S. Leslie, E. Eskin, and W. S. Noble, "The spectrum kernel: A string kernel for svm protein classification," *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, pp. 564–75, 2002.

[12] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2014.

[13] R. Torres, J. Kunkel, M. F. Dolz, and T. Ludwig, "A novel string representation and kernel function for the comparison of i/o access patterns," in *Parallel Computing Technologies*. Springer International Publishing, 2017, pp. 500–512.

[14] D. Fu, Y. Xu, H. Yu, and B. Yang, "Wastk: A weighted abstract syntax tree kernel method for source code plagiarism detection," *Scientific Programming*, 2017.

[15] "JPlag: Detecting software plagiarism," 2017. [Online]. Available: https://jplag.ipd.kit.edu/

[16] D. Gitchell and N. Tran, "Sim: A utility for detecting similarity in computer programs," *SIGCSE Bull.*, vol. 31, no. 1, pp. 266–270, Mar. 1999.

[17] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello, "A tree kernel based approach for clone detection," in *IEEE International Conference on Software Maintenance*, Sept 2010, pp. 1–5.

[18] U. Bandara and G. Wijayarathna, "Source code author identification with unsupervised feature learning," *Pattern Recognition Letters*, vol. 34, no. 3, pp. 330 – 334, 2013.

[19] P. Danphitsanuphan and T. Suwantada, "Code smell detecting tool and code smell-structure bug relationship," in *Engineering and Technology (S-CET), 2012 Spring Congress on*, 5 2012, pp. 1–5.

[20] S.-B. P. Hyun-Je Song and S. Y. Park, "Computation of program source code similarity by composition of parse tree and call graph," *Mathematical Problems in Engineering*, 2014.

[21] A. Sharma, A. K. Pujari, and K. K. Paliwal, "Intrusion detection using text processing techniques with a kernel based similarity measure," *Computers & Security*, vol. 26, no. 78, pp. 488 – 495, 2007.

[22] C. Wang, S. Hirasawa, H. Takizawa, and H. Kobayashi, "Identification and elimination of platform-specific code smells in high performance computing applications," *International Journal of Networking and Computing*, vol. 5, no. 1, pp. 180–199, 2015.