A Robust and Automatic Approach for Matching Algorithms

Alessandro Midolo¹, Emiliano Tramontana¹

¹Dipartimento di Matematica e Informatica, University of Catania, Catania, Italy

Abstract

Modern software applications can have millions of lines of code. Moreover, while several developers collaborate to develop code, each of them could have their own coding style, e.g. follow a personal code convention, prefer some data structures, etc. Having a large amount of code and several coding styles can negatively affect the readability of the code, especially when this has not been documented properly. Furthermore, a somewhat incorrect version of an algorithm would lead, besides to the presence of bugs, an involute, complex and inefficient code.

This paper proposes an automatic approach that aims at matching two (or more) versions of an algorithm. By means of static analysis, our approach finds the statements of the analysed algorithm, applies a transformation to avoid depending on some details, e.g. naming conventions, and then computes a similarity score between the resulting statement list and a set of known algorithms that have been collected and analysed beforehand. The similarity score will be computed according to a custom version of the Levenshtein distance, tailored to handle a statement list. The proposed approach has been tested on some sample algorithms to check the accuracy of the analysis and the precision of the match.

The said solution can assist the developer to ease the understanding of the source code, suggest improvements, and propose alternative versions, e.g. from recursive to iterative or viceversa, less memory-demanding, functionally correct, etc.

Keywords

Static analysis, coding suggestions, code reuse, refactoring

1. Introduction

The bigger the repositories the greater the demanding effort for developers when trying to understand code implemented by their colleagues. Visually inspecting source code to understand its structure and the functionalities could be time consuming and error-prone, since developers could misidentify some algorithms. Generally, this occurs when the source code documentation is poor, or missing, and when many developers contribute to the same project.

Automatic Program Comprehension (PC) tries to solve these issues by proposing several approaches that automatically assist developers to understand source code [1, 2]. Several applications of the PC have been presented: 4, 5], code clone detection, and algorithm recognition. Automatic approaches can help in analysing, comprehending and improving the source code, by means of tools that support the works of developers [6, 7, 8].

The state of the art shows several techniques to automatically identify algorithms. Machine learning approaches have been presented [9, 10], where different classifiers are used to label code fragments. Other proposals use a hybrid approach, mixing static analysis to extract data from the code and machine learning classifiers to identify the algorithm [11, 12, 13]; their classifier is based on a set of structural and truth value characteristics that are strictly related on sorting algorithms, therefore new beacons should be defined for different categories of algorithms. Furthermore, these approaches require a new training of the dataset if new labels are included in the classification, which can be time consuming and complex.

This paper focuses on algorithm recognition and presents an innovative approach that automatically matches algorithms by inspecting the source code. Our approach uses static analysis to collect data from the source code and compute a similarity score with templates of known algorithms to identify the correct one. The use of templates source code classification according to specific categories [3, guarantees that: new algorithms can be easily added for the recognition step; multiple versions of the same algorithm can be used to improve the accuracy of the identification; many, if not all, categories of algorithms can be recognised (sorting, searching, traversing, etc.). The proposed approach consists in four main phases: firstly, a code parsing tool collects all the statements of the algorithm analysed; secondly, a statement transformation is performed to extract the data required for the similarity match; thirdly, the Levenshtein distance is computed to attribute a similarity score between the algorithm and a set of known templates, representing other algorithms; finally, the template with the highest similarity score is selected.

> Levenshtein distance has been widely used in program analysis, especially for code clone detection when evaluating the similarity between code fragments [14, 15, 16].

SATToSE 2023: 15th Seminar on Advanced Techniques & Tools for Software Evolution, June 12-14, 2023, Fisciano (Salerno), Italy △ alessandro.midolo@phd.unict.it (A. Midolo);

tramontana@dmi.unict.it (E. Tramontana)

D 0000-0002-9575-8054 (A. Midolo); 0000-0002-7169-659X (E. Tramontana) © 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 Internet and 100 Provide

Commons License Attribution 4.0 International (CC BY 4.0). CEUR Workshop Proceedings (CEUR-WS.org)

However, such approaches are strictly related to detecting clone fragments. Such approaches focus on detecting type-3 clones that are portions of code that differ in terms of whitespaces, comments, layouts and identifiers, and can have some modifications like addition or removal of statements [17]. Conversely, we propose another approach in which different statements of the algorithms under analysis are reflected on the matching score, hence having a varying degree of matching; moreover, the above said approaches refer to types and names (e.g. methods and fields names) to detect clones, whereas our approach focuses on the statements freed from the developer chosen names, providing a more generalised identification.

The rest of the paper is organised as follows. Section 2 describes the proposed approach with all the steps followed by the analysis. Section 3 shows three examples of the use of the approach and how the similarity score is computed on real scenarios. Section 4 displays the metrics obtained by the analysis of the examples previously shown and a comparison with a text similarity approach. Section 6 reports meaningful related works and compares them with our approach. Finally, conclusions are drawn in Section 7.

2. Proposed Approach

We propose an approach that gathers data by parsing the source code and then evaluates the similarity score of an algorithm and a set of known algorithms. The proposed approach makes a proper generalisation of the algorithms to avoid depending on naming conventions or on statements that are not contributing to the main goal of the analysed algorithm. We make use of the static analysis of the source code, hence executable files are not needed for the analysis. Algorithm 1 shows the main steps, as psuedo-instructions, followed by the proposed approach to match algorithms. The procedure takes as input the source code, variable SC, and parses it extracting the compilation units. Then, method declarations are extracted and, for each, all the statements types are collected and compared to the algorithm templates to compute the similarity score. The approach can be structured in four steps: (i) parsing the source code by means of a tailored Visitor to gather all the data needed for the following analysis; (ii) selecting and transforming the most relevant statements; (iii) computing an adapted Levenshtein distance to determine the similarity score; (iv) evaluating the matching degree of the algorithms.

2.1. Code Parsing

We perform code parsing, by means of the Javaparser library, to extract all the data required to evaluate the simAlgorithm 1 The algorithm of the proposed approach

procedure MatchingAlgorithm(SC)
$compilationUnits \leftarrow parseAllPaths(SC)$
for cu, compilationUnits do
methods \leftarrow visitMethods(cu)
end for
for mDecl, methods do
$stmts \leftarrow mDecl.getStatementsType()$
for tmp, templates do
$Tstmts \leftarrow tmp.getStatementsType()$
$score \leftarrow computeLVD(stmts, Tstmts)$
mDecl.collectScore(tmp, score)
end for
end for
end procedure

ilarity score. Javaparser is an automatic parser that generates an abstract syntax tree (AST) from source code and provides a set of APIs to perform operations on it [18].

The root of the AST is the *CompilationUnit* (representing a Java file) to which all code elements are connected, e.g. package declaration, class and methods declarations, etc. Code inspection has been performed by using the *VoidVisitorAdapter* class, which lets us define a *Visitor* class to search for a specific property. In the *Visitor* class, the method *visit()* was implemented, which takes as parameters the type of object being searched (e.g. method declaration, statement, field), and the container in which data are stored; the body of the method contains all the instructions that are executed for every object found of the type specified as a parameter.

We have defined a *Visitor* which looks for *MDs* (method declarations); once a *MD* is found, the *visit()* method extracts from its body all the statements, and stores them in a *List* preserving the order. The list of statements provided by Javaparser is further expanded, as it would otherwise miss: (i) nested statements (e.g. all the statements present in the body of a *for* loop, defined as *ForStmt*), and (ii) expressions, such as assignments, method calls, variable declarations, etc. (defined as *ExpressionStmt*).

2.2. Statements Transformation

The statements initially gathered by Javaparser are transformed to better serve the following analysis. Javaparser provides a function, *getStatements()*, to get the statements contained in the body of a method declaration; nested statements, e.g. statements contained in a for loop, are omitted be the said *getStatements()*, and just the parent statement is inserted in the list. To collect all the statements inside the method, we further extract nested statements and we place them in the list of statements right after their parent, preserving the order of the block. Whereupon, for each statement in the list, we extract the

Table 1

Some examples of statement types defined in the Javaparser library (see the documentation for the complete list¹). The *ExpressionsStmt* is handled differently from others, since it can represent more types of expressions (method calls, assignments, declarations, etc.).

Statement Type	Code
BreakStmt	break;
ContinueStmt	continue;
DoStmt	$do\{\ldots\}$ while $(a > 0);$
ExpressionStmt	*
ForEachStmt	for(Object : objects){}
ForStmt	$for(a = 3; a < 99; a + +)\{\}$
lfStmt	$if(a == 0)\{\}$
ReturnStmt	return a;

class type, avoiding collecting other parts such as names, types, comments and expressions, to better generalise the approach, so as to recognise different versions of the same algorithm; e.g. statement for(int i=0; i<size; i++) is represented in Javaparser by a ForStmt type, whereas all the other parts, such as the variable declaration (int i=0), the binary expression (*i*<*size*) and the unary expression (i++) are omitted. For all the statements that can contain nested statements in their body, e.g. for, if, while, a custom statement is inserted at the end of the nested statements; its name is given by the concatenation of the "End" prefix with the type of the statement containing the nested ones, e.g. EndIf, EndFor, EndWhile. This custom statement allows us to identify which statements are nested, thus improving the precision of the approach when comparing algorithms.

The extracted types are defined by the Javaparser library¹. Table 1 shows some examples of statement types: the column *Statement Type* represents the type defined by the library, and the column *Code* shows an example of the code associated with the type; further types can be found in the documentation. The *ExpressionStmt* type does not have a code example because it can represent any type of expression: *AssignExpr* (a = b + c;), *MethodCallExpr* (*method()*;), *VariableDeclarationExpr* (*int* a = 0;) etc. This class type is too generic, hence we select and insert in the statement list the type of the expressions, e.g. a method call with argument an *ObjectCreationExpr* (e.g. *method(a, new object())*), we select the parent expression, in this example the *MethodCallExpr*.

Listing 1 shows an example of data extracted from a method: the code on top shows the method *bubbleSort()*; the bottom part displays the list of statements extracted by our visitor. The statement list extracted by Javaparser

contains just a *ForStmt* because all the other instructions are nested into it, whereas our approach has properly handled this occurrence and the statement list is defined as follows: the first two instructions are *for* statements, the third is a *if* statement, the fourth is a variable declaration expression and the last two are assign expressions.

Statements Type: ForStmt, ForStmt, IfStmt, VariableDeclarationExpr, AssignExpr, AssignExpr EndIf EndFor EndFor EndFor

Listing 1: The upper part shows an iterative version of the *bubblesort* algorithm implemented in Java, whereas the bottom part displays the list of statements extracted by our approach, using the class types defined in Javaparser.

2.3. Adapting the Levenshtein Distance

The Levenshtein distance is a string metric for measuring the difference between two sequences [19]. It is defined as the minimum number of operations (replace, insert and delete) required to change a sequence into the other. A string can be seen as a list of single characters; the Levenshtein algorithm iteratively compares all the characters and finds the minimum number of operations (insertion, deletion or substitution) required to make the two sequences equal. We have implemented a custom version of the algorithm where two lists of statements are the compared sequences, and every character represents a single statement. Once defined the number of minimum instructions, the similarity score is computed as [20]:

$$Similarity(S_1, S_2) = 1 - \frac{levDist(S_1, S_2)}{max(size(S_1), size(S_2))}$$

 $^{^1\}mbox{https://www.javadoc.io/doc/com.github.javaparser/javaparsercore/latest/index.html}$

where S_1 and S_1 are the two sequences of statements, *levDist()* gives as output the Levenshtein distance, and *size()* gives the number of elements in a sequence.

2.4. Algorithm Recognition

Once all the data needed for the analysis have been obtained, we can compute the similarity scores according to the extracted list of statements, i.e. the analysed method is compared with a set of known algorithms that have been gathered and parsed beforehand. We have created a set of Java files containing the source code of several known algorithms, and for each one at least two versions are stored: iterative and recursive. For some algorithms, more versions have been implemented as the aim is to improve the accuracy of our analysis. E.g., the bubblesort algorithm has two different versions, besides the iterative and recursive versions: the one shown in Listing 1, and an optimised version where a boolean flag breaks the execution if no elements are swapped in the inner loop.

Listing 2 shows one of the templates of the iterative version for the bubblesort algorithm used in our analysis: the code on top displays the implementation of the algorithm, while the list below represents the statements extracted by our approach. We show this template because, according to our approach, it is the most similar to the code shown in Listing 1. However, the two methods have several textual differences: firstly, the name of some variables is different, e.g. the variable representing number of elements, *size* and *length*, the array containing the elements, *sort_arr* and *list*, and the variable used for the swap, *tmp* and *swap*; secondly, the condition in the *lfStmt* is inverted; finally, the template has an additional statement compared to the example, the first statement *VariableDeclarationExpr*.

Despite these differences, our approach correctly identifies the algorithm implemented. According to the Levenshtein distance, the number of operations needed to match the two sequences is one (an insertion because the list of statements differ in only one element). Indeed the similarity score between these two sequences is computed as *Similarity* = 1 - (1/7) = 0.857, where the the size of the longest sequence is 7.

3. Evaluation

We tested our approach on four different algorithms, each implemented as a method. All the templates used by our approach can be found on a public repository². The first example is shown in Listing 1 previously discussed; here, we discuss three other algorithms: a version of *factorial* and two versions of *quicksort*.

Statements Type: VariableDelcarationExpr, ForStmt, IfStmt, VariableDeclarationExpr, AssignExpr, AssignExpr EndIf EndFor EndFor

Listing 2: The upper part shows one iterative version of the *bubblesort* algorithm stored in our template db, whereas the bottom part displays the list of statements extracted by our approach, according to the types defined in Javaparser.

Listing 3 shows a method implementing the factorial algorithm for an integer value in a recursive form. The list of statements is as follows: *IfStmt, ReturnStmt, ReturnStmt.* The analysis carried out by our approach has identified the method as the recursive version of the factorial algorithm with a similarity of *1.0.* In such a case, the similarity is the maximum possible value since given the simple structure of the algorithm, the types of statements used by such a method match 100% the factorial algorithm template.

```
public static int factorial(int n) {
    if (n == 0 || n == 1)
        return 1;
    return n * factorial(n - 1);
}
```

Listing 3: An example of the recursive factorial algorithm implemented in Java.

Moreover, two versions of the quicksort algorithm are

²https://github.com/AleMidolo/MatchingAlgorithms

considered to test the approach on different versions of the same algorithm; both versions propose an iterative solution.

Firstly, Listing 4 shows an iterative version of quicksort algorithm that uses a stack as a support to sort the elements contained in the array passed as argument. The method consists in twelve statements, in order: two VariableDeclaration, MethodCall, WhileStmt, VariableDeclaration, AssignExpr, MethodCall, VariableDeclaration, and two IfStmt with a MethodCall in their body. Therefore, the method is characterised by such statements, indeed comments, names and types will be ignored for the purpose of the identification. Our template storage includes an implementation of the quicksort using a stack to sort the elements³; the differences between the template and this method are: the template has an additional VariableDeclarationExpression before the first push() call; in the example, the first instruction after the WhileStmt is a VariableDeclarationExpr, while in the template it is an AssignExpr; the partition() method call takes one more argument in the template; types and names of the variables are different. Considering the said differences, our analysis has computed a similarity score of 0.85, correctly identifying the algorithm.

Secondly, Listing 5 displays an iterative version of the quicksort algorithm that uses a supporting array to sort the elements of the array passed as argument. The method consists in fifteen statements, in order: three VariableDeclaration, two AssignExpr, WhileStmt, three AssignExpr, and two IfStmt with two AssignExpr in their body. There are two main differences in the structure between the two versions of the same algorithm: the total number of statements, twelve against fifteen, and the absence of MethodCall statements in the second version. An iterative version of the quicksort algorithm is stored in our template storage, and it uses an array to sort the elements like the method given as an example. The main differences between the template and the method are the following: the method has three VariableDeclarationExpr before the WhileStmt, whereas the template has only two; types and names of variables are different. The analysis has computed a similarity score of 0.86.

Our approach correctly identified both versions because the analysis uses templates for different versions of the same algorithm, making the recognition more accurate. Still, the storage containing all templates can be updated with more versions of algorithms to make the approach more sensitive to differences and up-to-date.

```
public void quickSortStack(short[] array) {
  // create a stack for storing
  // subarray start and end index
  Stack<Pair> st = new Stack<>();
  short finish = array.length -1;
  // push the start and end index
  // of the array into the stack
  st.push(new Pair(0, finish));
  // loop till stack is empty
  while (!s.empty()) {
    // remove top pair from the list and get
    // subarray starting and ending indices
    short begin = st.peek().getX();
    finish = st.peek().getY();
    st.pop();
    // rearrange elements across pivot
    short pv = partition(array, begin);
    // push subarray indices with elements
    // less than the current pivot to stack
    if (pv - 1 > begin) {
        st.push(new Pair(begin, pv - 1));
    }
    // push subarray indices with elements
    // more than the current pivot to stack
    if (pv + 1 < begin) {
        st.push(new Pair(pv + 1, begin));
    }
  }
}
```

Listing 4: An iterative version of the quicksort algorithm using a stack of objects to sort the elements.

4. Results

We have compared our approach to a text-based search approach between methods and the templates of the algorithms. Table 2 shows the metrics obtained for the four methods previously described: column method displays the method considered for the analysis, respectively bubblesortV1 (listing 1), factorialV1 (listing 3), quicksort-Stack (listing 4) and quicksortArray (listing 5). The other five main columns are the templates used by our approach to match the algorithms: *ItBubblesort* is the bubblesort iterative version (the one displayed in Listing 2); RecFactorial is the factorial recursive version; ItQuicksortST is the quicksort iterative version using a stack to sort the elements; *ItOuicksortAr* is the quicksort iterative version using an array to sort the elements; ItMergesort is the mergesort iterative version. We have also considered the mergesort to show how the analysis is able to distinguish different algorithms. Each of these columns have two subcolumns: sim and text are respectively the similarity score of our approach and the similarity score of the text

³The templates in the db can be found in the github repository given above.

Table 2

Similarities between the examples shown before and five different known algorithms used by our approach as templates. The first column, *method*, shows the name of method analysed, while the other columns displays for each template the similarity score given by our approach, column *sim*, and by a text comparison approach, column *text*.

	ItBubblesort		RecFactorial		ltQuicksortSt		ItQuicksortAr		ItMergesort	
method	sim	text	sim	text	sim	text	sim	text	sim	text
bubblesortV1	0.85	0.52	0.16	0.05	0.15	0.28	0.21	0.30	0.16	0.21
factorialV1	0.14	0.07	1.0	0.7	0.07	0.09	0.07	0.05	0.06	0.04
quicksortStack	0.25	0.21	0.08	0.06	0.84	0.37	0.42	0.29	0.23	0.26
quicksortArray	0.26	0.21	0.06	0.02	0.46	0.28	0.86	0.34	0.30	0.33

public static void quickSortArray(

```
long arr[], long low, long high) {
// Create an auxiliary stack
long[] list = new long[high - low + 1];
long max = -1;
long pivot = 0;
// push initial values to stack
list[++max] = low;
list[++max] = high;
// Keep popping from stack while not empty
while (max >= 0) {
 h = list[max--];
  1 = list[max--];
  // Set pivot element at its correct
  // position in sorted array
 pivot = partition(arr, low, high);
 // If there are elements on left side
  // of pivot, push left side to stack
  if (pivot - 1 > low) {
      list[++max] = low;
      list[++max] = pivot - 1;
  }
  // If there are elements on right side
  // of pivot, push right side to stack
  if (pivot + 1 < high) {
      list[++max] = pivot + 1;
      list[++max] = high;
  }
}
```

Listing 5: An iterative version of the quicksort algorithm using an array to handle the sort of the elements.

comparison approach.

}

We have highlighted in bold text the highest score corresponding to the correct identification of the algorithm in subcolumn *sim*; whereas in subcolumn *text* the maximum score has been highlighted for the text match. We can see that the score assigned by our approach is much higher in each case, indicating a higher precision in the identification.

5. Discussion

For all the five methods shown in Table 2, our approach shows a higher identification score compared to the text similarity approach. Indeed, we have a higher similarity score that is more than double for the *quickSortStack* (0.84 compared to 0.37) and *quickSortArray* (0.86 and 0.34) methods, and values about 40% greater for *bubblesortV1* (0.85 and 0.52) and *factorialV1* (1.0 and 0.7) methods. Our approach performs better because it can generalise the matching, without considering names of variables, comments and names of types.

Moreover, the matching scores given by our analysis are clearly greater than the score of other algorithms, whereas, with a text similarity approach, we can see that the quicksortArray method has a similarity score of 0.34 for *ItQuickSortAr*, 0.33 for *ItMergesort* and 0.28 for *ItQuicksortSt*, hence the closeness of such scores can bring ambiguity in the correct identification of the algorithm.

Finally, our approach can distinctly recognise two different versions of the same algorithm: the *quicksortStack* has 0.84 score as *ItQuicksortSt* and 0.42 as *ItQuickSortAr*, while the *quicksortArray* method has respectively 0.46 and 0.86. The accurate identification of the version used is crucial when suggesting improvements or proposing different versions.

The ability to recognise an algorithm is related to the set of templates, which is not easy to maintain and grow. An algorithm can be matched if a similar template of the same algorithm is already part of the templates. To handle this, the database can be populated with the most popular algorithms, and, if an algorithm is not included, it could be added by a developer using our approach and tool.

The approach extracts the statement's type to evaluate the similarity between algorithms. On the one hand, two algorithms can have similar statements despite having a different behaviour, thus showing low accuracy in identifying code's behaviour. On the other hand, the approach can give a degree of generalisation, since it is not dependent on names and types encountered, therefore it is not based on the comprehension of how the algorithm was implemented, but on its structure. This property is the main difference between our approach and type-3 clone approaches.

6. Related Works

Algorithm recognition has been tackled by several approaches in the state of the art, both for the software engineering industry and for academic settings. In [9], the authors present a solution for automatic algorithm recognition using machine-learning techniques; they extract a dataset of source code containing algorithms, then a feature extraction is carried out to collect all the characteristic data (e.g. count-vars, operators, constructs etc.), finally a tag updating is done to remove redundant tags. They have trained the dataset and built a group of classifiers to identify the algorithm. In order to add a new algorithm or category of algorithms to the classification, all the previous steps have to be executed again in order to properly train the new dataset, which it could be time consuming; conversely, since our approach uses static analysis to match templates, a new template can be added to the collection without the need for further operations.

In [13], the authors propose an algorithm recognition method that detects sorting algorithmic schemas; these schemas consist in a set of loops, features, operations etc. Another approach discussing sorting algorithms is presented in [12, 11], where numerical (number of blocks, number of loops etc.) and descriptive (iterative, recursive) characteristics are extracted from the source code and a C4.5 decision tree classifier is builded to detect sorting algorithms. These approaches focus on sorting algorithms under some assumptions such as algorithms are expected to be implemented in a well-established way, e.g. quicksort algorithm should be implemented in a recursive way since it is more common. Moreover, they define a set of characteristics that are mostly related to sorting algorithms. In contrast, our approach can identify a wider spectrum of algorithms since the static analysis can be performed to any Java source code, and we consider different versions of the same algorithm to increase the identification ability.

Many tools have been presented to measure source code similarity. Most of these approaches address problems such as code clone detection, software licensing violation and software plagiarism [21]. The Levenshtein distance is often used for clone detection. In [16], a hybrid technique is presented where source code is lexically analysed to detect and extract sub-blocks, then similar blocks are grouped and hashed, finally the *Levenshtein similarity* and the *Cosine Similarity* are used to compute similarity between blocks and find Type-3 clones. In [14], the authors propose a cross-language clone detector for C, C++ and Java, the input code is tokenized to obtain the keywords of the corresponding language, then these keywords are compared with the Levenshtein distance and finally the clone types are classified based on similarity of keywords match. In [15], the authors present a tool to detect clones of a faulty code fragment, a Normalised Compression Distance is defined to detect duplicate code fragments.

The above said approaches use the Levenshtein distance to detect code clone fragments, whereas our proposal defines a tailored distance to match methods with algorithm templates in order to achieve algorithm recognition. Furthermore, these tools are sensitive to differences in term of statements between code fragments, because the presence or absence of multiple statements can lead to a misidentification of a code similarity; whereas our approach proposes a similarity score which, despite the statements differences, is able to define a matching grade for all templates, where the highest one is the most similar to the implemented algorithm.

7. Conclusion

This paper presented an automatic approach to recognise algorithms using static analysis. By parsing the source code it is possible to identify all the statements composing a method, transform them according to a specified format, then compute Levenshtein distance for obtaining a similarity score between the method and several templates of known algorithms. The template having the highest score is suggested as the algorithm matching the analysed method. We performed an experiment on four methods to test our approach; the results obtained highlight a high accuracy when recognising the algorithm compared to a textual similarity.

The versatility of the approach allows us to add more templates to widen the spectrum of recognisable algorithms and to increase the number of different versions of algorithms. This approach can be employed both for program comprehension purposes on software development, supporting developers in understanding and implementing source code, by proposing alternative versions of the same algorithm, and for academic purposes to automatically assess students' assignments.

Acknowledgments

The authors acknowledge project TEAMS funded by University of Catania PIACERI 2020/22.

References

- [1] J. Siegmund, Program comprehension: Past, present, and future, in: Proceedings of 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 5, 2016, pp. 13–20. doi:10.1109/SANER.2016.35.
- [2] W. Maalej, R. Tiarks, T. Roehm, R. Koschke, On the comprehension of program comprehension, ACM Transactions on Software Engineering Methodology 23 (2014). doi:10.1145/2622669.
- [3] S. Ugurel, R. Krovetz, C. L. Giles, What's the code? automatic classification of source code archives, in: Proceedings of 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2002, p. 632–638. doi:10.1145/775047. 775141.
- [4] K. Tian, M. Revelle, D. Poshyvanyk, Using latent dirichlet allocation for automatic categorization of software, in: Proceedings of 6th IEEE International Working Conference on Mining Software Repositories (MSR), 2009, pp. 163–166. doi:10.1109/MSR. 2009.5069496.
- [5] S. Srikant, V. Aggarwal, Automatic grading of computer programs: A machine learning approach, in: Proceedings of 12th International Conference on Machine Learning and Applications (ICMLA), volume 1, 2013, pp. 85–92. doi:10.1109/ICMLA.2013. 22.
- [6] A. Midolo, E. Tramontana, Refactoring java loops to streams automatically, in: Proceedings of 4th International Conference on Computer Science and Software Engineering (CSSE), 2021, p. 135–139. doi:10.1145/3494885.3494910.
- [7] A. Midolo, E. Tramontana, An api for analysing and classifying data dependence in view of parallelism, in: Proceedings of 10th International Conference on Computer and Communications Management (ICCCM), 2022, p. 61–67. doi:10.1145/ 3556223.3556232.
- [8] A. Midolo, E. Tramontana, Automatic generation of accurate test templates based on junit asserts, in: Proceedings of the 7th International Conference on Algorithms, Computing and Systems (ICACS), 2023.
- [9] M. Shalaby, T. Mehrez, A. El Mougy, K. Abdulnasser, A. Al-Safty, Automatic algorithm recognition of source-code using machine learning, in: Proceedings of 16th IEEE International Conference on Machine Learning and Applications (ICMLA), 2017, pp. 170–177. doi:10.1109/ICMLA.2017.00033.
- [10] W. Crichton, G. G. Sampaio, P. Hanrahan, Automating program structure classification, in: Proceedings of 52nd ACM Technical Symposium on Computer Science Education (SIGCSE), 2021, p.

1177-1183. doi:10.1145/3408877.3432358.

- [11] A. Taherkhani, L. Malmi, A. Korhonen, Algorithm recognition by static analysis and its application in students' submissions assessment, in: Proceedings of 8th ACM International Conference on Computing Education Research (ICER), 2008, p. 88–91. doi:10.1145/1595356.1595372.
- [12] A. Taherkhani, Recognizing sorting algorithms with the c4.5 decision tree classifier, in: Proceedings of 18th IEEE International Conference on Program Comprehension (ICPC), 2010, pp. 72–75. doi:10.1109/ICPC.2010.11.
- [13] A. Taherkhani, L. Malmi, Beacon-and schema-based method for recognizing algorithms from students' source code, Journal of Educational Data Mining 5 (2013) 69–101. doi:10.5281/zenodo.3554635.
- [14] S. B. Ankali, L. Parthiban, Development of porting analyzer to search cross-language code clones using levenshtein distance, in: Proocedings of Fourth International Conference on Smart Computing and Informatics (SCI), Springer, 2021, pp. 623–632. doi:10.1007/978-981-16-0878-0_60.
- [15] T. Ishio, N. Maeda, K. Shibuya, K. Inoue, Cloned buggy code detection in practice using normalized compression distance, in: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 591–594. doi:10.1109/ICSME.2018.00022.
- [16] A. Sheneamer, J. Kalita, Code clone detection using coarse and fine-grained hybrid approaches, in: Proceedings of 7th IEEE International Conference on Intelligent Computing and Information Systems (ICICIS), 2015, pp. 472–480. doi:10.1109/ IntelCIS.2015.7397263.
- [17] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, B. Maqbool, A systematic review on code clone detection, IEEE Access 7 (2019) 86121–86144. doi:10.1109/ACCESS.2019.2918202.
- [18] N. Smith, D. Van Bruggen, F. Tomassetti, Javaparser: visited, Leanpub, oct. de (2017).
- [19] V. I. Levenshtein, et al., Binary codes capable of correcting deletions, insertions, and reversals, in: Soviet physics doklady, volume 10, Soviet Union, 1966, pp. 707–710.
- [20] A. Niewiarowski, et al., Short text similarity algorithm based on the edit distance and thesaurus, Czasopismo Techniczne 2016 (2016) 159–173. doi:10. 4467/2353737XCT.16.149.5760.
- [21] C. Ragkhitwetsagul, J. Krinke, D. Clark, A comparison of code similarity analysers, Empirical Software Engineering 23 (2018) 2464–2519. doi:10. 1007/s10664-017-9564-7.