Clone detection in student programs based on lexical analysis of source codes

O N Polovikova¹ and V E Ivanova¹

¹ Institute of Mathematics and Information Technology, Altai State University, 61 Lenina ave., Barnaul, 656049, Russia

E-mail: ponOlgap@gmail.com, fairytaill@mail.ru

Abstract. This article presents the development of automatic clone detection for verification of students programs - the Detector of Clones. The Levenshtein Distance is used for the initial assessment of the similarity of the two token sequences which is calculated by the Wagner-Fisher algorithm. Verification of two programs is carried out by the fingerprint method and by the winnowing method. To improve the results of plagiarism assessment, an additional calculation was added to the implementation of the winnowing method: the detector compares not only the labels themselves, but also the sequences near the matching labels. The detector of clones is formed for the cycle of disciplines Object Oriented Programming, Object Oriented Analysis and Design on the Moodle platform. These tools support C-like languages (C++ and C#). In general, the work of the detector undoubtedly made it possible to increase the consistency in verification of student programs.

1. Introduction

The availability of information, including in the form of source code, leads to the borrowing of ideas and data in various forms of electronic representation. The Internet space is not only an information source for completing assignments and research work, but also an interface for copying them directly without attribution. The information and educational environment of the university involves structured data banks, as well as temporary repositories of student work, which can also be sources for borrowing.

By means of messengers and social networks, students not only exchange news, but also the results of their research studies, copying each other's thoughts, texts, and program codes. The problem of data borrowing in the educational space is considered in the works of Russian and foreign authors [1-4]. Scientific and educational communities have become interested in copying data in research, articles and programs. Today, the automated systems with the function for verification for plagiarism of both text works and program code are relevant [4].

Various services have developed and using to detect of clones the text works (article A.O. Siebert and V.I. Khrustaleva offers an overview of the principles of operation of modern Internet services for verification texts for borrowing [3]), and for the recognition of plagiarism of program code at the moment there is no adequate software, and therefore this topic still remains relevant [5, p. 135]. The main differences and the difficulties associated with them are that the program code is naturally similar in syntax, since different people can use the same programming language constructs. Therefore, a simple verification for word matching will not give adequate results [4].
This problem is especially acute for teachers of disciplines in which students are required to write program code to perform typical tasks [6, p. 418]. The volume of work being done and the limited time do not allow a teacher to evaluate the uniqueness of the code, and objectively evaluate the completed task. Federal Requirements for Educational Standards of Russia (FSES-3) in the direction “Information systems and technologies (09.03.02) include general professional competencies that can be obtained through practical experience of writing software blocks and modules independently. For example, there is the General Professional Competence «Able to develop algorithms and programs suitable for practical use in the field of information systems and technologies». Thus, studies on the problem of borrowing program code in a number of disciplines in which students must write program modules independently are especially important for organizing a full-fledged educational process.

Undoubtedly, automatic (or semi-automatic) code borrowing verification is in demand not only for the learning process itself (a graduation work, an ongoing laboratory work), but also for organizing and conducting Olympiads, competitions and programming tournaments [5]. In this case, as in the evaluation of laboratory work, it is required to detect in short time, including uniqueness, a large amount of program code. The time limit requires taking into account the complexity of the technology of such verification.

The accumulated experience (methods and technologies) of the fight against computer piracy (obfuscation, authentication, hashing and control flow control and others [7]) in this context is practically useless for several reasons. The author of the program code should be interested in copy protection, which in most cases does not take place for various reasons. Code manipulations, such as the adding decision-making operators to the code for authentication, the adding digital watermarks, the encoding using keys, or the hiding the structure of components of a software system, require additional technologies, certain skills from the authors, and additional time to complete. But, most importantly, such resource-intensive and time-consuming manipulations to protect programs are not advisable for the current program modules for laboratory or olympiad tasks.

As part of the organization of the educational process, teachers can use the existing developments of code plagiarism testing (a review and testing of such developments is presented in the work of Yu. Lifshits [8]). It should be noted that in most cases, the applied methods are tied to a specific programming language, the proposed functionality does not allow accumulating a bank of works for subsequent comparison [6, p. 420], a long-term adjustment of metrics and parameters by means of analyzing the results by numerous experiments is necessary.

The specificity of the task of checking for uniqueness of the code itself, as well as the features of the specific disciplines that need it, do not allow (at this stage of research in this direction) talking about the need to develop a universal detector (an automated system for checking for plagiarism). Universality often leads to time-consuming tuning of characteristics and parameters, and in order to get the required accuracy of verification, we should understand the details of the methods used and their adaptation to the subject area. Such a «universality» fee is commensurate with the new development of a detector for a particular discipline or several disciplines.

It should be understood that highly specialized detectors, for example, to extract borrowings by one programming language, to verification only one type of duplicate code fragments (type of code modifications) [9, 10], to work only one assessment method, not able to answer the modern needs and requirements of Education System.

2. Creating the detector of clones based on token representation
The first of all needs to form the concept of requirements according to which the logic of detector operation will be programmed (figure 1):

1. Two-stage check. Initial evaluation of the program by algorithm with low computational complexity (class P). Subsequent evaluation by a method that is resistant to the first and second type of duplicate code fragments.

2. Selection of program comparison mode: with one (optional) or/and with several components from some collection.
3. Viewing results with highlighting (color marking) «suspicious» blocks.
4. Generation of the report in several formats: for the student, for the teacher.

![Figure 1. The use case diagram for the Detector of Clones.](image)

The experience of testing programs and models performed by students, as well as the analysis of studies on these issues, allowed concluding that in order to detect clones, it is necessary «to teach» the Detector to identify the first three types of duplicate code fragments. The fourth type, associated with a change in the structure of the program block, is difficult to execute (implement). Within the framework of solving typical problems Fourth type cannot be unambiguously classified as plagiarism (borrowing). In this case, both program blocks can be implemented on the basis of this classic example, with different adaptations. Thus, the system under development should interpret code blocks with different formatting parameters in the same way, ignore comments and find the same blocks even if some operator is intentionally added and/or deleted.

Since the result of the detector operation is the numeric grade of the program codes, then the principles and methods of the system should ensure calculation of certain numerical values. By numeric grade the decision-maker can conclude that the work is unique with respect to some collection of works. In essence, the detector must program the calculation of some the proximity function. The result of which will be a numerical criterion for evaluating the program in the case of a «one-to-one» comparison mode. For the «one-to-many» mode of operation, it is proposed to generate and display a descending-sorted list of the proximity function results that are greater than a certain threshold value.

Despite the variety of approaches used for such comparison, they can be divided into four main groups according to the level of data structures, on the basis of which the program code is analyzed: text, lexical, syntax, semantic.

*Text*-based clone detection methods consider source code as a collection of lines or characters. In this case, no additional data structures or representations are used. Codes can be compared by combining several methods, for example, a partial or complete match of the maximum number of lines is determined, or the maximum length of equal character blocks is determined, etc. Obviously, such methods are not able to find duplicate code fragments of the first and second type [10] (not find blocks of code that are formatted differently).

The *lexical* approach is based on the use of token sequences [6]. In essence, tokenization is the encoding of program lines in terms of a special tokenization scheme. The analysis of the program code is replaced by the analysis of the tokenized representation. A very important role is played by the tokenization scheme. By changing the classes of tokenization scheme, we can affect the operation of the detector. The larger the number of classes of tokens, the more detailed the program lines are.
encoded, since the tokens of one class are represented by one value. Replacing operators and declarations on tokens for the source code, this is an attempt to abstract from the implementation details and highlight only the characteristic structure of this program. In the tokenized code, there is no specification of names and identifiers, which means that such a representation is resistant to changes in the code by such modifications.

The methods of the syntactic approach are based on the analysis of syntactic Trees and Graphs [11] generated for programs. In this case, a variety of search algorithms are used on Trees and Graphs, on the basis of which not only detectors of clones work, but also various error analyzers and translators. Despite the variety of template implementations of algorithms on Graphs and Trees, storing such structures in a matrix representation (adjacency matrix) is a resource-intensive process [12], even if adjacency lists are used for the case of discharged Graphs.

A semantic analysis approach works with a Program Dependency Graph (PDG) — this is a combined data flow Graph and control flow Graph. The tops of the PDG are the instructions of the program, and the edges are the dependencies between instructions of the program. This approach has high computational complexity. In research Sevak Sargsyan and other authors [13], it was shown that the search for code clones reduces to the calculation of isomorphic Subgraphs, the complexity of such a calculation can have a cubic dependence on the number of tops in the PDG.

In works on the problems under consideration, an approach based on the use of metrics is highlighted. In systems, metric-based algorithms are typically implemented in conjunction with textual, lexical, parsing, or parsing algorithms. Despite the obviousness of evaluating the «proximity» of programs based on the «proximity» of metrics, such methods only provide an initial estimate due to the low accuracy in comparison with methods of other approaches.

Undoubtedly, the most correct form of representing program code for performing comparisons in order to identify clones is syntactic or semantic Graphs (Trees). But, the resource-intensiveness of the graph storage process and the high computational complexity of the search algorithms objectively do not allow using these approaches for a detector that should work in the «one-to-many» mode. In this study, a lexical approach is used to determine the «proximity» of codes. Thus, clones code blocks will use «relatively equal» chains of tokens sequential. In this case, the order of the semantic units of the language is important (building ads, operator, calls, etc.), as well as what elements of the language are used.

Let us single out the stages of obtaining an estimate by the detector in the «one-to-one» comparison mode:

1) downloading files with the source code of programs as an answer to the task in an electronic educational course on the Moodle platform;
2) preparation of program code;
3) getting tokens sequential;
4) calculating the Levenshtein Distance (primary criterion) for comparison with the programs selected from the collection by the teacher, depending on the number of the task;
5) allocation of suspicious sections of two programs (based on tokens sequential) by the winnowing algorithm of the fingerprint method;
6) calculation of the proximity function using the allocated hash codes.

When we download the program to the Moodle platform, a new record is added to the MySQL database. A set of scripts, that are physically located in the same space as the files under study perform preliminary code processing and tokenization of all downloaded programs. The results of the tokenization (tokens sequential) are stored in the corresponding MySQL DBMS tables. The work of the detector is carried out by scripts that are already executed on the side of the MySQL DBMS. The interface to the detector functions is built into the corresponding electronic course on the Moodle educational platform.

To prepare for the tokenization stages, in system has implemented an additional stage - code preparation (transferring all characters to lower case, deleting non-essential characters. At the tokenization stage, each operator is assigned a certain code, depending from the operator class (each
language has own tokenization scheme). Then, the tokens sequences are formed. The tokenization process in the program acts as a separate block (the class that implements these steps is presented in figure 2) and can work independently of other parts of the program. Input - textual representation of the input source code of the program. Output is a dynamic string of tokens.

![Figure 2. Class implementing the tokenization mechanism (C#).](image)

For the initial assessment of the similarity of the two tokenized representations, the Levenshtein Distance is used [14], which is calculated by the Wagner-Fisher algorithm [15]. To select the metric of the initial evaluation, testing was conducted on 57 student programs corresponding to one task. Three metrics participated in the testing: the Jacquard Distance, the Levenshtein Distance, and the Kolmogorov Distance. The reliability of the results performed by the Levenshtein Distance was the highest on this sample of programs. This is also confirmed by the study of I.A. Posov and V.E. Dopir [16, p. 65].

The Levenshtein Distance between two lines (tokenized representation) is the minimum number of operations to insert one character, delete one character and replace one character with another, necessary to turn one line into another (see the recurrence formula [17, p. 61]). Thus, the «proximity» of two tokenized representations is calculated as the minimum amount of the operation of replacing, inserting, and deleting a character necessary to convert one string to another. The simplicity and high accuracy of the results by the Levenshtein Distance (in comparison with other metrics) made it possible to define this metric as the primary criterion for Detector of Clones.

\[
d(S_1, S_2) = D(M,M),
\]

where

\[
D(i,j) = \min \left\{ \begin{array}{ll}
0 & ; i = 0, j = 0 \\
| \vphantom{i} & \\
i & ; j = 0, i > 0 \\
| \vphantom{j} & \\
D(i-1,j-1) & ; i = 0, j > 0 \\
| \vphantom{i} & \\
D(i,j-1) + \text{insertCost} & ; j > 0, i > 0, S_1[i] = S_2[j] \\
| \vphantom{i} & \\
D(i-1,j) + \text{deleteCost} & \\
| \vphantom{j} & \\
D(i-1,j-1) + \text{replaceCost} & ; j > 0, i > 0, S_1[i] \neq S_2[j]
\end{array} \right. 
\]

(1)

where \(\min(a,b,c)\) – function returning the smallest value;
the line items are numbered from the first; \(S_1\) and \(S_2\) – two lines (length \(M\) and \(N\) respectively) in some alphabet, where \(d(S_1, S_2)\) is the Levenshtein Distance.

After calculations by the primary criterion, the proximity function (return percentage) between the two programs is calculated using the fingerprint method [18, p. 191] based on the winnowing algorithm [18, p. 192]. A significant moment in the implementation of the fingerprint method is the choice of the
number $k$, which is responsible for limiting the length of the smallest substring that this algorithm can work with. For each tokenized representation, substrings of length $k$ are hashed; rows with a length shorter than $k$ are not considered. Based on the received hash codes, the algorithm (the winnowing algorithm is used) selects and marks a subset of them that characterizes the given representation well, and then writes the marked areas to special tables. For marked areas from different representations (different programs), an intersection is sought, the length of which determines the percentage of proximity between the two programs. Based on the intersections of the marked areas, the proximity function (percentage of proximity of the two codes) is calculated.

Accordingly, for small values of $k$ (for example, 2 or 3), the detector will consider that the blocks of program with several definitions or operators are suspicious. In this case, there will be many false suspicions. But, if we take a value for $k$ much larger than the average string length in the tokenized representation, in this case there is a high probability to «missing» plagiarism. The number $k$ for the system (detector) under consideration is calculated dynamically based on an analysis of the line lengths of the file. Empirically, they came to the conclusion that the static restrictions on the length of the smallest substring do not allow one version of the detector to evaluate different types of programs. Object-oriented code with a long series of service words and the code of programs with functions of templates have significantly different average length of strings of tokenized representation.

Selected by algorithm hash-codes are stored in a special database table. This allows the detector to work in the «one-with-many» plagiarism verification mode without repeated transformations and constructions of hash-codes sets characterizing the program. Figure 3 shows the class template by the winnowing algorithm for the fingerprint method.

```java
public class Fingerprints {
    private List<Integer> fingerprint = new List<Integer>();
    private List<Integer> FingerprintPosition = new List<Integer>();
    public List<Integer> Fingerprint {
        ...
    }
    public List<Integer> FingerprintPosition{
        ...
    }
    static private bool control = true;
    /* Ring hash function to hash tokenized substrings*/
    private Int64 RingHash(string str, int pos, int lex, Int64 prevHash){
        ...
    }
    /* Screening the winnowing algorithm implementation */
    void winnow(Int64[] hash){
        ...
    }
    public Fingerprints(string Token, int lexemSize, int winSize) {
        Int64[] hash = new Int64[Token.Length - lexemSize + 1];
        for(int i = 0; i < Token.Length-lexemSize+1; ++i)
            if(i == 0)
                hash[i] = RingHash(Token, 1, lexemSize, 0);
            else
                hash[i] = RingHash(Token, i, lexemSize, hash[i - 1]);
            winnow(winSize, hash);
    }
}
```

**Figure 3.** Class for implementing the winnowing algorithm.

When changing the copied code, system warnings and hidden code errors remain in the program, because to correct such shortcomings, we need to fully analyze and rebuild the program code. To identify such a flaw, specialized programs are used – static analyzers of the source code of the program. The analyzer allows detecting various code defects and generate log files with a report. Thus, the analyzer report is also a reliable «fingerprint» of the program, which reflects its uniqueness.
For all downloaded as an answer to the task of program codes in the Moodle environment, an automatic analysis is performed, a report is generated. Two types of analyzers are used (for C++ and C#), but types of analyzers were not disclosed to students. This motivates students to study supporting tools for programming. By acquiring testing skills with such programs, there is an understanding of the need for a detailed study of the subject area to build high-quality code. The work of analyzers complicates the rash use of modifications with code when copying. Primitive changes to introduce «noise» into the operation of detector algorithms (declaration of additional variables, operators and methods) will be reflected in the reports of the analyzers. Therefore, the task of increasing the stability of the developed detector to the third type of duplicate code fragments has been partially solved.

3. Conclusion
In addition, to improve the result, an additional calculation was added to the implementation of the winnowing algorithm: the detector compares not only the labels themselves, but also the tokenized representation blocks near the matching labels. To implement this action, the idea of «tagged elements» was formulated for the greedy string tiling algorithm [19]. Testing has shown the adequacy of such a change. The assumption that next to the chain of copied statements and definitions there may be another copied code chain has a logical justification, since borrowing is performed blocks. For example, the student independently encodes input and output, and the necessary code for the transformations (program filling) copies and applies various methods of code modifications.

As part of the study, the Detector of Codes was developed for a bank of programs. The program bank is formed for the cycle of disciplines Object-Oriented Programming, Object-Oriented Analysis and Design on the Moodle platform. At this stage, the detector supports C-like languages (C++ and C#), operates in the comparison mode «one-with-one» and «one-with-many» (figure 4 shows an example of the detector’s report). Using only a characteristic chain of hash-codes of the tokenized representation to calculate the percentage of proximity of two program files allowed organizing a comparison in the «one-with-many» mode. Indexed storage of such chains in a segmental database allows plagiarism codes to be estimated for an acceptable time, even with an increase in the number of samples.

| file 1 | file 2 | Levenshtein distance | Similarity percentage | The percentage of coincidence of the first program with the second | The percentage of coincidence of the second program with the first |
|-------|-------|----------------------|-----------------------|---------------------------------------------------------------|---------------------------------------------------------------|
| A-482-1 | E-484-1 | 297                  | 50,80%                | 23,70%                                                        | 23,24%                                                        |
| A-482-1 | P-485-2 | 605                  | 25,67%                | 7,11%                                                         | 2,23%                                                         |
| A-482-1 | E-485-5 | 388                  | 43,97%                | 28,50%                                                        | 28,63%                                                        |

| file 1 | file 2 | Levenshtein distance | Similarity percentage | The percentage of coincidence of the first program with the second | The percentage of coincidence of the second program with the first |
|-------|-------|----------------------|-----------------------|---------------------------------------------------------------|---------------------------------------------------------------|
| P-454-12-2 | X-451-12-2 | 186                  | 61,09%                | 15,96%                                                        | 18,83%                                                        |
| P-454-12-2 | X-451-12-2 | 422                  | 24,51%                | 5,94%                                                         | 8,41%                                                         |
| P-454-12-2 | Φ-451-1 | 190                  | 72,94%                | 78,25%                                                        | 76,24%                                                        |
| P-454-12-2 | Φ-454-2 | 210                  | 55,41%                | 62,92%                                                        | 89,38%                                                        |
| P-454-12-2 | Φ-454-1-4 | 561                  | 22,78%                | 12,13%                                                        | 8,51%                                                         |
| P-454-12-2 | P-454-1-4 | 313                  | 39,46%                | 9,94%                                                         | 12,38%                                                        |
| P-454-12-2 | Φ-455-6-1 | 637                  | 22,88%                | 6,94%                                                         | 9,64%                                                         |
| P-454-12-2 | Γ-455-2-10 | 327                  | 47,95%                | 22,16%                                                        | 31,15%                                                        |

**Figure 4.** Version of the report on the detector (for a teacher).
In general, the work of the Detector of Codes undoubtedly made it possible to increase the consistency of the teacher's work in checking and objective evaluation of student programs. At the same time, it is necessary to agree with the opinion of research teachers on this issue that the work of the detector can significantly reduce the time required to check student programs, increase the objectivity of the grades. But we must not forget that the final decision on each work should be made by the teacher (decision maker). Since the detector’s assessment is only advisory in nature.

References
[1] Chen X, Francia B, Li M, McKinnon B and Seker A 2004 IEEE Trans. Information Theory 1545-1550
[2] Hattingh F G, Buitendag A A K and Van der Walt J S 2013 Journal of Information Technology Education: Innovations in Practice 12 45-58
[3] Siebert A O and Khrustalev V I 2014 Universum: technical sciences 3(4) 5-14.
[4] Strelchenok G V, Lepikhin T A and Labzo K S 2015 Modern Information Technologies and IT Education 11 211-214
[5] Malyshev E V and Smelov V V 2018 Transactions of BSTU: Physics and mathematics and computer science 1(206) 135-138
[6] Shikov A N and Sorokin D S 2014 General Relativity 2 418-427
[7] Sarwar S, Ul Qayyum Z, Safyan M, Iqbal M and Mahmood Y 2019 Trudy ISP RA /Proc. ISP RAS 31.2 171-186 DOI: 10.15514/ISPRAS-2019-31(2)-12
[8] Lifshits Yu Overview of automatic plagiarism detectors in programs (SPb: GUIT MIO)
[9] Chanchal K R, James R C and Koschke R 2009 Science of Computer Programming 74(7) 470-495
[10] Bellon S, Koschke R, Antoniol G, Krinke J and Merlo E 2007 IEEE Transactions on software engineering 33(9) 577-591
[11] Srinuvasu M A 2016 International Journal of computer sciences and engineering 2(7) 157-168
[12] Toivonen H 2011 Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining 965–973
[13] Sargsyan S, Kurmangaleev S, Baloiyan A and Aslanyan H 2014 Mathematical Problems of Computer Science 42 54-62
[14] Levenshtein V I 1965 Reports of the USSR Academy of Sciences 163.4 845-848
[15] Wagner R A and Fischer M J 1974 Journal of ACM 21(1) 168-173
[16] Posov I A and Dopira B E 2019 Izvestiya SPbGETU «LETI» 6 61-66
[17] Leshchenko A V 20018 Collection of scientific papers of NSTU 3-4(93) 59–69 DOI: 10.17212 / 2307-6879-2018-3-4-59-69
[18] Evtifeeva O A, Crass A L, Lakunin M A, Schastlivtsev R R and Lysenko E A 2007 Sci. and Techn. Journal of Information Technologies, Mechanics and Optics 5 188-196
[19] Wise M J 1993 Basser Technical report, Basser Department of Computer Science, University of Sydney, Australia