Program clones detection as natural language texts fragments based on constructive-synthesizing modeling

Viktor Shynkarenko[1,*,†], Olena Kuropiatnyk[1,†] and Inna Demidovich [1,†]

[1] *Ukrainian State University of Science and Technologies, academician Lazaryan 2, 49010 Dnipro, Ukraine*

Abstract

The developed and tested method for comparing the structure of natural language texts is adapted to the analysis of program texts. The method is based on the use of stochastic grammars, including rules that describe the algorithmic structure of programs. The certain structures appearance probability is calculated as the product of the different program elements probabilities. Constructive-production modeling tools were used to form the rules. An experiment was conducted to verify the possibility of using this method to detect clones in the programs source text in C++ and C#. Different types of tasks and their software implementations were studied: both those that are equivalent in control flow but different in calculations, and vice versa. As a result of the experiments, it was found that programs that solve different tasks but have almost identical algorithms have high values of similarity indicators. If the algorithms are similar, but solve different tasks, the indicators are slightly lower. Similarity indicators from low to medium, obtained in cases where different tasks are solved with different algorithms that is due to the use of a single programming language syntax.

Keywords

software, constructive-synthesizing modeling, natural language, formal language, program clone, information technology

1. Introduction

Similar or identical fragments of code can occur in the programs texts. They are called code clones [1], which are in some sense analogous to borrowings in natural language texts. The presence of clones reduces readability and complicates the programs debugging process and their technical support. The task of detecting clones can be attributed to code refactoring. Due to the possible large volumes of program texts, the need to search clones in different modules and versions of programs, it is advisable to develop and use automated tools for detecting clones.

Currently, some of software tools based on them for detecting clones already exist. They are based on approaches inherent in processing texts in natural (text, token) and formal languages (abstract syntax trees and their modifications, graphs, etc.) [2] [3]. Accordingly, each has its own advantages and disadvantages.

The object of research in this work is the clones detecting processes in program texts. The subject of the study is a method for detecting such clones based on stochastic grammars. It has satisfactory performance working with natural language texts [6] [7] and was adapted to formal languages within the framework of this study.

Scientific novelty: for the first time, a method for detecting clones based on stochastic grammars constructed using constructive-synthesizing modeling tools has been proposed. The existing

CLW-2025: Computational Linguistics Workshop at 9th International Conference on Computational Linguistics and Intelligent Systems (CoLInS-2025), May 15–16, 2025, Kharkiv, Ukraine

[*] Corresponding author.

[†] These authors contributed equally.

shinkarenko_vi@ua.fm (V. Shynkarenko); olena.kuropiatnyk@ust.edu.ua (O. Kuropiatnyk); 2019demidovichinn@gmail.com (I. Demidovich)

0000-0001-8738-7225 (V. Shynkarenko); 0000-0003-2286-884X (O. Kuropiatnyk); 0000-0002-3644-184X (I. Demidovich)

© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-3976/paper2.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

classification of clones has been supplemented, namely, by the detection of the known (fourth) type inversion: of syntactically similar, but semantically different code fragments.

The practical significance is in the clone detection automatization by using software that implements the proposed method. The software can later be used in code refactoring and plagiarism detection tasks.

2. Related works

In the texts of programs, functionally equivalent or syntactically similar fragments can be found that are clones. An example of functional equivalence is the implementation of various array sorting algorithms. Syntactically similar ones include implementations of algorithms for finding the maximum and minimum element of an array.

The concept of a clone is related to the fragments semantics and syntax. Thus, in [8] code clones are semantically similar pairs of code fragments that may have syntactic similarity. Semantic similarity is determined by the functional identity or similarity of program fragments, and syntactic similarity is determined by the similarity of the program text [8] [9]. The purpose of searching for clones is to detect fragments with similar functionality. This can simplify program maintenance [1] [8], warning [8] and bug detection [1] [10], copyright plagiarism detection [1], refactoring [1] [11] [12].

In the general taxonomy [8] [9] [13], [14], four types of clones are distinguished based on differences at the syntactic or semantic level: 1-3 syntactically similar, 4 – semantically similar:

1) syntactically identical code fragments, except for differences in whitespace and comments;

2) in addition to the previous one, syntactically identical code fragments, except for differences in identifier names and literal values;

3) in addition to the previous two, syntactically similar code fragments that differ at the expression level. Expressions can be added to these fragments, changed and/or deleted relative to each other;

4) syntactically dissimilar code fragments that have the same functionality. For example, one fragment that implements bubble sort and another fragment that implements quick sort are considered a type 4 clone of code.

Also, sometimes additional types are distinguished – structural and functional clones [14]. The first of them follow from the syntactic structure of a particular programming language and are observed within the structure or class. The last of them are limited to detail at the procedure or method/function level. Both can be of any of the types 1 – 4.

Clone detection methods are divided into five main classes [9]: textual, token, syntactic (tree-based, metric-based), semantic, and machine learning-based. The last of them are the least studied. A subclass of semantic methods can be considered those that use the Program Dependency Graph (PDG). PDG includes control and data flows.

In the textual approach, two code fragments are compared to each other in the form of text/lines/tokens and are found to be cloned only if the two code fragments are literally identical in textual content. A little more flexibility is provided by the use of regular expressions [11].

In the token-based approach, all lines of code are divided into a sequence of tokens during the lexical analysis of the compiler. All tokens are then converted back into token sequence strings, which are matched to identify and report clone code. This approach is used in CCFinder [13] to detect type 1 and type 2 clones. SourcererCC [13] captures the similarity of overlapping tokens to detect type 3 clones and can be scaled for large code (up to 250 million lines of code). In [15], a modification of token-based clone detection is proposed that allows detection of more clone pairs with greater diversity without losing accuracy by performing the search multiple times, targeting different groups of clones.

However, due to the lack of application semantics consideration, these token-based approaches cannot handle type 4 clones (semantic clones).

Syntactic methods include those that use syntax trees. Abstract Syntactic Trees (AST) with Augmented Flow (FA-AST) are used to search for semantic clones [8]. In this case, the original AST is supplemented with information about the types of edges: child, parent, next sibling, next token, next use. Several additional types of edges are also used to represent the control flow of the program: sequential execution, If statements, While and For loops. Then, two different types of graph neural networks are applied to the FA-AST to measure the similarity of code pairs. To improve the accuracy of the results obtained based on AST and tokens, an approach based on code stylometry matcher is proposed [16].

To determine semantic clones, approaches based on graphs and their isomorphism are also used, but they cannot scale to large code due to the complexity of graph isomorphism and the high time required to compare graphs. The control flow graph is used as the graph. In the following, the graph is considered as a social network, to which the method of social network centrality analysis is applied [13]. According to this method, the "centrality" for each token in the basic block and the total centrality of this token in different basic blocks are allocated. As a result, we obtain semantic tokens – certain lexemes with graph details. To determine the similarity of code pairs, a Siamese neural network is used. A similar approach was used in the development of the TreeCen application, which showed high efficiency compared to analogues [17].

A control flow graph approach with token and edge criticality annotation and Siamese network analysis was used in [18] for parallel clone detection.

In [19], semantic code clone detection using program dependency graphs and geometric neural networks is proposed, using structured syntactic and semantic information. Based on this approach, a prototype of the Holmes tool was developed and its performance was empirically evaluated.

In [10], an approach to clone detection is proposed based on the control flow analysis and dominator trees, which represent all possible paths to reach a certain point in a program. Dominator trees are used to compare structural fragments of code. The similarity of the fragments dominator trees may indicate that their functions are similar and that they may be potential candidates for structural clones or subclones – smaller parts of code that have similar structure or functionality to another one, larger part of the code.

In [1], a PDG-based code cloning detector, CCGraph, is proposed, which uses graph kernels. The PDG structure is normalized and a two-step filtering strategy is developed by measuring the characteristic vectors of the codes. Then, code clones are detected using an approximate graph matching algorithm based on the reformation graph kernel WL (Weisfeiler-Lehman).

Thus, we can conclude that text and token methods are simple to implement, but sensitive to changes in the order of tokens in the program text, and are unable to detect clones of types 3 and 4.

Syntactic and semantic methods are based on the use of tree and graph structures, that is, they strictly depend on the programming language. Building structures is time-consuming, and graph matching is a resource-intensive operation for large amounts of data.

Machine learning methods require additional data sets for training.

Regarding experimental estimates of the clone identification accuracy and the time spent on it, one cannot give preference to tools based on natural or formal language processing methods [5].

In our work, we propose a method for assessing program similarity based on constructive-productive modeling and stochastic grammars, which can be used to detect functional clones. The work considers clones belonging to types 1-3, as well as the "inverted" fourth: syntactically similar, but semantically different code fragments.

3. Methods

To construct the rules of stochastic grammar and their further analysis, constructive-synthesizing modeling is used [6]. These rules are a model of the program algorithmic structure. The probability of application in the program is determined for each rule. The probability of deriving the entire program is determined as the product of the certain algorithmic structures appearance probabilities with different levels of nesting.

Constructive-synthesizing modeling is based on the use of a generalized constructor and its refining transformations.

A generalized constructor C_G is the triple $C_G=\langle M,\Sigma,\Lambda\rangle$, where M is heterogeneous carrier that contains terminals and non-terminals and can be expanded during the construction process; Σ are operations and relations signature, consisting of following operations: binding, substitution and derivation, operations with attributes and the substitution relation; Λ is information support of construction (ISC).

According to Λ form $_{w}l$ with w attribute is called a set of terminals and non-terminals, which are combined by binding operations. In the developed constructors, a single binding operation (and relation) is used and called concatenation. A construction is a form that contains only terminals. The formation of constructions occurs by deriving from the initial non-terminal, performing substitution operations and operations on attributes and generalized operations of partial and full derivation.

The partial derivation operation consists of selecting the appropriate substitution rule from their set, performing this substitution and performing operations on attributes that correspond to the selected rule in a certain sequence.

The operation of full derivation (or simply derivation) consists of the partial derivation operation sequential execution, starting from the initial nonterminal and ending with the construction.

To form constructions, it is necessary to perform a number of clarifying transformations with constructor:

- specialization – determines the subject area of construction: the semantic nature of the carrier, a finite set of operations and their semantics, operations attributes, the order of their execution and restrictions on substitution rules;
- interpretation – consists in linking signature operations with algorithms for executing a certain executor to form a certain construction;
- concretization – expansion of the axiomatics with a set of production rules, setting specific sets of nonterminal and terminal symbols with their attributes and, if necessary, attribute values;
- implementation – construction formation from elements of the constructor carrier by executing algorithms related to signature operations.

As a result of C_G refinement transformations (specialization, interpretation, and concretization), three constructors were formed: a constructor-transformer that transforms program text into tagged text, a constructor-transformer of tagged text into a set of formal substitution rules with a probability measure, and a constructor-measurer of the two program texts similarity degree [6] [7] [20].

The effectiveness of the proposed method was investigated for solving the problems of establishing the fictional and technical texts similarity degree [9, 10, 18]. In this study, the described method is adapted for texts written in a formal language, namely program texts. The purpose of comparing texts is to identify similar ones for refactoring, namely, to remove program clones.

The description of the program structure is performed using reserved words and identifiers, which the program is divided into, and their certain sequence or nesting in each other, for example: a loop with a conditional statement inside.

Among the reserved words in the program text, the following tags were used: loop, branch, process. After that, the data type, reserved word or variable name, operator type and nesting level are specified for each tag as its attributes. Attributes used: function (func), class (class), variable (var), operator (op), data type (type), number (num), symbol (sym).

For each algorithmic structure, its appearance probability in a certain place of the program text is calculated [20]. After receiving the program text in the form of tags sequences set with their appearance probability in a specific place, rules are formed. The probability is calculated as the number of occurrences in the text divided by their total number.

In experiment 1, the constructors form the rules of stochastic grammar obtained as a result of processing programs that solve tasks 1 and 2.

Task 1: find the maximum value of an element in a two-dimensional array. Input: two-dimensional array, array dimension. Output: the value of the maximum element in the array. The listing is below.

```
public static int MaxValue(int[,] array)
{
int max = array[0, 0];
for (int i = 0; i < array.GetLength(0); i++)
 for (int j = 0; j < array.GetLength(1); j++)
{
if (array[i, j] > max)
max = array[i, j];
}
 return max;
}
```

Task 2: sort array elements using the bubble method. Input: one-dimensional array, array dimension. Output: ordered one-dimensional array. The listing looks like this:

```
public static void BubbleSort2(int[] array)
{
for (int i = 0; i < array.GetLength(0); i++)
  for (int j = 0; j < array.GetLength(0) - 1 - i; j++)
  {
if (array[j] > array[j + 1])
       {
int tmp = array[j];
     array[j] = array[j + 1];
array[j + 1] = tmp;
   }
  }
}
```

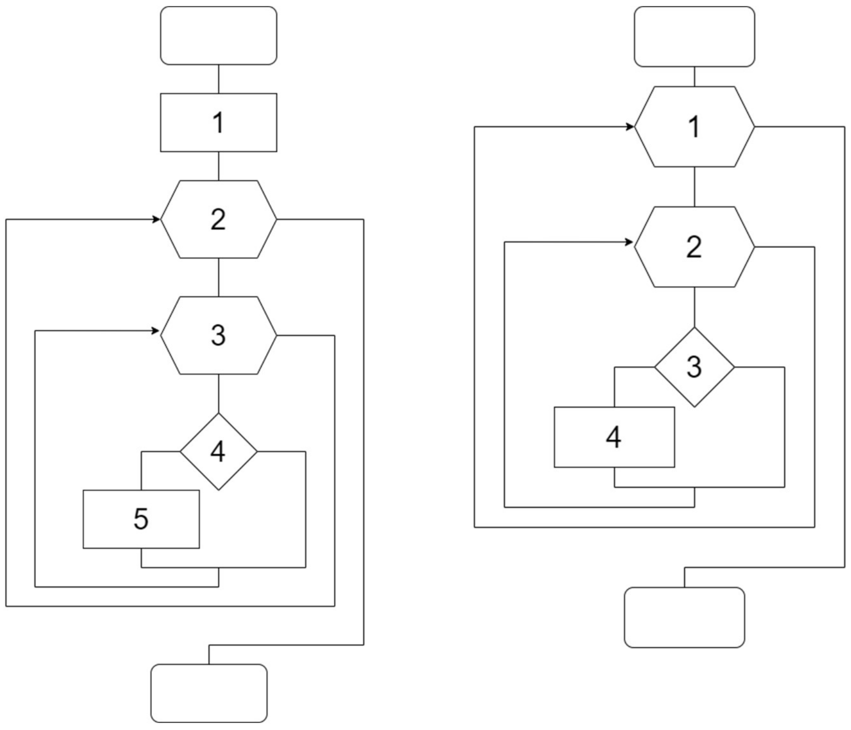
These programs have different functionality. The syntax similarity is partly due to the use of the C# language in both cases. However, the greatest influence on the similarity of the tokens order is played by the implemented algorithmic structures and their sequence, that is, the structure of the program algorithm.

For its analysis, it is possible to switch to one of the representations: a control graph or an algorithm scheme in the form of a flowchart.

In this work, we will use the last one, since a flowchart is a graph with a vertices type definition (determined in accordance with the algorithmic structure being represented) and, compared to the control graph, carries more information.

Let us consider the solutions to problems 1 and 2 in terms of algorithmic structures: sequence, selection, repetition.

The fig. 1 presents structural diagrams of algorithms for solving tasks 1 and 2.



a b

Figure 1: Generalized scheme of the algorithm: a – to task 1, b – to task 2

Let us analyze the qualitative and quantitative composition of the algorithms for solving problems 1 and 2 (Table 1).

Table 1.
The fig. 1 schemes structure comparison in terms of different blocks amount

	Task 1	Task 2
Process block	2	1
Choice	1	1
Repetition	2	2

To implement each of the tasks, process blocks, choice and repetition are used (Table 1). Blocks 2-5 correspond in type and mutual placement to blocks 1-4 of task 2 (Fig. 1).

Thus, two functionally different programs of tasks 1 and 2 have a structural difference only in linear block 1.

From the asymptotic computational complexity point of view, the algorithms and their implementations are equal (*$O(n^2)$*), that is, the differences in the code do not cause significant changes in the computational complexity. That is why moving on to the direct analysis, we will take into account the operations of assignment, address transition and comparison.

The computational complexity indicators are in the worst case for *MaxValue = $7n^2+5n+3$*, *BubbleSort2 = $5n^2+5n+4$*. The difference is *$2n^2-1$* in favor of *MaxValue*.

In this case, we can conclude that the clones have the same asymptotic computational complexity. The differences between them has a complexity of *$O(n^2)$*.

As an example, there are the rules of stochastic grammar that model the program for the task 2:

$$\sigma_1 \xrightarrow{\text{p1,1}} loopA_{1,1};A_{1,1} \xrightarrow{\text{p2,1}} typeA_{2,1};A_{2,1} \xrightarrow{\text{p3,1}} varA_{3,1};$$

$$A_{3,1} \xrightarrow{\text{p4,1}} opA_{4,1};A_{4,1} \xrightarrow{\text{p5,1}} numA_{5,1};A_{5,1} \xrightarrow{\text{p6,1}} varA_{6,1};$$

$$A_{6,1} \xrightarrow{\text{p7,1}} opA_{7,1};\ A_{7,1} \xrightarrow{\text{p8,1}} funA_{8,1};A_{8,1} \xrightarrow{\text{p9,1}} varA_{9,1};A_{9,1} \xrightarrow{\text{p10,1}} opA_{10,1}.$$

$$\sigma_2 \xrightarrow{p1,2} loopA_{1,2}; A_{1,2} \xrightarrow{p2,2} typeA_{2,2}; A_{2,2} \xrightarrow{p3,2} varA_{3,2};$$
$$A_{3,2} \xrightarrow{p4,2} opA_{4,2}; A_{4,2} \xrightarrow{p5,2} numA_{5,2};$$
$$A_{5,2} \xrightarrow{p6,2} varA_{6,2}; A_{6,2} \xrightarrow{p7,2} opA_{7,2};\ A_{7,2} \xrightarrow{p8,2} funA_{8,2}; A_{8,2}$$
$$\xrightarrow{p9,2} opA_{9,2}; A_{9,2} \xrightarrow{p10,2} numA_{10,2}; A_{10,2} \xrightarrow{p11,2} opA_{11,2}; A_{11,2} \xrightarrow{p12,2} varA_{12,2};$$
$$A_{12,2} \xrightarrow{p13,2} varA_{13,2}; A_{13,2} \xrightarrow{p14,2} opA_{14,2}.$$

$$\sigma_3 \xrightarrow{p1,3} branchA_{1,3};\ A_{1,3} \xrightarrow{p2,3} varA_{2,3}; A_{2,3} \xrightarrow{p3,3} opA_{3,3};\ A_{3,3} \xrightarrow{p4,3} varA_{4,3}.$$
$$\sigma_4 \xrightarrow{p1,4} procA_{1,4}; A_{1,4} \xrightarrow{p2,4} typehA_{2,4}; A_{2,4} \xrightarrow{p3,4} varA_{3,4}; A_{3,4} \xrightarrow{p4,4} opA_{4,4}; A_{4,4} \xrightarrow{p5,4} varA_{5,4}.$$

$$\sigma_5 \xrightarrow{p1,5} procA_{1,5}; A_{1,5} \xrightarrow{p2,5} varA_{2,5}; A_{2,5} \xrightarrow{p3,5} opA_{3,5}; A_{3,5} \xrightarrow{p4,5} varA_{4,5}.$$
$$\sigma_6 \xrightarrow{p1,6} procA_{1,6}; A_{1,6} \xrightarrow{p2,6} varA_{2,6}; A_{2,6} \xrightarrow{p3,6} opA_{3,6}; A_{3,6} \xrightarrow{p4,6} varA_{4,6},$$

where σ is the initial non-terminal, $A_{i,j}, p_{i,j}$ are the j-th non-terminal and the probability in the i-th level rule. The level corresponds to the operator sequence number of the program being processed. The terminals are tags used for identifying reserved words as well as operator types and variables in the program (such as op for example '=', var for example 'temp', num for example '10' , etc.).

4. Results

According to the comparison results of the obtained rules (experiment 1), the similarity of the code for tasks 1 and 2 is 96% due to the identity of the logical structures used to solve these tasks. This allows programs 1 and 2 to be considered clone programs of type 4. In addition, the 3rd program was investigated, which, like the 2nd one, performed sorting, but in the 3rd program code, the `Length` field was used instead of the `array.GetLength()` function. Thus, when comparing all three programs, the following results were obtained: the result of comparing the second and third programs was 0.98, the first and third – 0.95.

The second half of experiment 1 the software code that solves the following tasks were compared:

- task 4: insert a new element into a single-linked one-way ordered list without breaking its order;
- task 5: insert an element into a single-linked one-way list into a given position using a While loop;
- task 6: insert an element into a single-linked one-way list into a given position using a For loop.

The obtained results of the software code comparison that solves each of these tasks are shown in the table below.

According to the obtained results (Table 2), the programs are functionally different, but structurally and syntactically similar – an inverted 4 type of clone. 1-3 and 4-6, show high similarity separately in the middle of each group – from 93% to 98%. While those that solve different tasks and do not have a similar algorithm in general received a level of coincidence from 14% to 24%, respectively.

In experiment 2, the similarity of 4 different programs solving the same task, but having a different code structure, was investigated. Two similar groups 7-10 (Table 3) and 11-14 (Table 4) were selected for the experiment.

Table 2
Degree of similarity of programs 1-6, %

	1	2	3	4	5	6
1	1	0.96	0.95	0.23	0.24	0.24
2	0.96	1	0.98	0.23	0.17	0.15
3	0.95	0.98	1	0.16	0.16	0.14
4	0.23	0.23	0.16	1	0.93	0.96
5	0.24	0.17	0.16	0.93	1	0.94
6	0.24	0.15	0.14	0.96	0.94	1

All programs within a group solve one task, different from another group. The obtained results of comparing the similarity of programs inside the group are given in the Tables 3-4.

Table 3
Degree of similarity of programs 7-10, %

	7	8	9	10
7	1	0.85	0.87	0.78
8	0.85	1	0.67	0.82
9	0.87	0.67	1	0.74
10	0.78	0.82	0.74	1

Table 4
Similarity degree for programs 11-14, %

	11	12	13	14
11	1	0.67	0.76	0.74
12	0.67	1	0.89	0.85
13	0.76	0.89	1	0.88
14	0.74	0.85	0.88	1

As can be seen in the Table 5, for programs with different algorithms, but written to solve the same task, the similarity in the middle of the group is somewhat lower: from 67% to 87% in the first group and from 67% to 89% in the second group, respectively.

On average, the degree of programs similarity that do not have a structurally similar algorithm and do not solve a common problem, the degree of similarity reached from 14% to 60%. A fairly high degree of programs similarity that are not clones is due to the use of similar, limited by programming language, tools.

To obtain a more informative result, students' programs were selected that solved a certain problem with a total number of 200 files.

The first half of them solved the problem of processing a matrix of integers using structures and functions.

Table 5
Summary table of results, %

Group №	Types of programs in the group	Similarity Percentage
1	different tasks but similar algorithm, programs 1-3	95-98
2	same data structure different task, programs 4-6	93-96
3	same task different data structure, programs 7-10	67-87
4	same task different data structure, programs 11-14	67-89
5.1	working with matrix, different tasks	73-80
5.2	working with matrix, same tasks	85-94

6.1	working with classes, different tasks	65-70
6.2	working with classes, same task	75-78
7	group 1 and 2 comparison	14-24
8	group 3 and 4 comparison	14-60
9	group 5 and 6 comparison	20-35

The following requirements were put forward for the program code:

- represent the matrix as a structure (struct);
- represent each new action as a function. The following set of functions is mandatory:
    - main function of the program;
    - creation of a matrix (indicating the dimension, memory allocating);
    - clearing the memory occupied by the matrix;
    - matrix processing should not contain input and/or output operators;
    - filling the matrix from the keyboard and using a random number generator, respectively;
    - displaying the matrix on the screen;
- provide the possibility of entering the matrix dimension and choosing the method of filling it (from the keyboard, using a random number generator);
- comment on the matrix processing function(s), based on the logical subtasks into which the main task is divided;
- it is forbidden to use auxiliary arrays and matrices, including STL containers.

The content of the processing function depends on the individual task variant. In total 10 different task variants are provided.

The obtained results of 100 programs comparison demonstrated their overall similarity at the level of 73-94%, where the similarity level of programs written for different variants was 73-80%, and when the variants coincided (the programs solved the same task and differed only in the programming style of the performer), the percentage reached 85-94%.

Each of the second half programs solved the task of creating a class that models an entity according to the variant of the individual task, and demonstrating work with its objects in the main function of the program.

Each class must have: at least three attributes; three types of constructors (default, with parameters and copying); destructor; setters, getters; toString method, which returns a string containing the names and values of all attributes. Also, for a given class, it is necessary to implement a friendly entity: a function or class according to the variant, and overload three given operations.

The comparison results reached 65-87% of the total similarity, with the similarity of programs solving different variants being 65-70% and for the common variant 75-87% of the similarity, which is mostly due to the use of common tools by different performers with their own style of writing program code.

When comparing programs from the first and second half with each other, their similarity varied at the level of 20-35% percent, which indicates the inevitable coincidence in a certain part of the programs, provided that they were written in the same programming language due to the use of tools limited by the capabilities and syntax of that programming language.

5. Discussion

The greatest similarity was found when comparing programs with a structurally similar algorithm for solving the task. However, it should be noted that a common algorithm does not mean a common task, so when comparing programs 1 and 2, which solve different problems but have almost identical algorithms (Fig. 1), the obtained similarity reached 96%.

Programs that have similar algorithms, but solve different problems, according to conducted studies, have a similarity of 67% to 89%. And programs that do not have a common task and a similar algorithm, due to the use of a common programming language toolkit, have a similarity from 14% to 60%.

The results show that the similarity of program texts is significantly higher within the same group, especially if the tasks have common algorithmic approaches or data structures (groups 1 and 2).

The results (Table 5, groups 1-2) show that the structural similarity of the algorithms can be identified as a semantic clone. This indicates that evaluating programs from the uniqueness point of view, it is appropriate to use functionality comparison based on the software testing methods.

Programs with the same tasks, but different data structures, have a lower level of similarity (groups 3 and 4), which may be due to the variety of implementations and the individual approach of each person to writing programs – their own style.

For work with matrices or classes, the similarity of tasks significantly affects the level of similarity (groups 6 and 8 show a higher level of similarity than groups 5 and 7).

The results of the fifth and sixth groups show the difference between the same and different tasks that in percentage terms does not exceed 14%. In practice, these results should be taken into account for assessing the educational work of education seekers.

Low similarity between different groups (significantly lower percentages of similarity in groups 9-11) indicate that different groups of programs have significant differences in implementation, even if the task or structure is partially similar.

The obtained results confirm that the main factors influencing the similarity of program texts are the number and sequence of algorithmic structures, data structure and similarity of tasks. The highest level of similarity is observed in programs that have the same algorithms or data structures.

However, the comparison between different groups demonstrates significant variability in the code, which indicates wide opportunities for creative implementation of tasks, while maintaining a certain minimum similarity of the program code at the level of 14-20% indicates the existence of basic elements and common structures in the code that are typical for a programming language or solving a certain type of tasks. This is due to the use of general principles and standards of programming, even in the case of different approaches to solving tasks.

The set of detected clones requires further analysis, especially how many fragmented parts make up the similarity in the detected percentage of clones (duplications) and whether they are logically complete parts that are subject to refactoring. It is also advisable to filter the detected clones by type, which will allow assessing the effectiveness of the proposed method for known clones types.

Based on the results obtained, the advantages and disadvantages of detecting software clones methods were identified. A number of criteria were proposed for comparing the methods. Data on compliance with the criteria are given in Table 6.

Due to the relevance of such investigations, the number of methods and their modifications is constantly growing, the data in Table 6 can be clarified and supplemented over the time. Also, methods for detecting clones based on machine learning are not taken into account.

Table 6
Comparison of methods for detecting clones

Criterion \ Method	Textual	Token	Syntactic	Semantic	Proposed method
Types of detected clone	1	1-2, 3*	1-4**	1-4	1-3, 4***
Availability of customization	-	+	-	-	+
Vulnerability to	+	-	-	-	-

permutations without change in content					
Scalability	+	+	+	-	+

The notes:

* SourcererCC [5] captures overlapping token similarity for type 3 clone detection;

** flow-augmented abstract syntax trees (FA-AST) are used to find semantic clones [2];

*** method detects clones of the fourth inverted type.

6. Conclusions

The paper proposes an approach to detecting clones based on stochastic grammars that describe the algorithmic structure of the program. To formalize the approach, the apparatus of constructive-synthesizing modeling is used.

The method based on constructive-synthesizing modeling with the formation of stochastic grammar rules was previously applied to determining the degree of similarity for fiction and technical texts in natural language. In this paper, its adaptation is performed to identify programs clone.

Conducted direct experiments made it possible to identify the similarity of program fragments based on the presented approach. The greatest similarity was found for fragments with a similar algorithmic structure. However, the similarity of the latter does not guarantee the functional similarity or the programs identity. This made it possible to define a new type of clones, which is an inversion of the known (fourth) type: syntactically similar, but semantically different code fragments.

In general, the practical significance of the proposed method and its software implementation is to provide a tool for comparing programs in order to detect clones and assess possible expected results of the uniqueness of program texts.

The method can be used to solve the following problems: code refactoring; programming pattern extraction; testing the uniqueness of programming training papers.

The detection of a new type of clones provides grounds for studying existing methods in order to assess their ability to detect such clones.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] Y. Zou, B. Ban, Y. Xue, and Y. Xu, CCGraph: a PDG-based code clone detector with approximate graph matching, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 931–942. doi:10.1145/3324884.3416541

[2] H. Yang, Z. Li, X. A Guo, Novel Source Code Clone Detection Method Based on Dual-GCN and IVHFS. Electronics, 12, 1315 (2023). doi:10.3390/electronics12061315

[3] G. Shobha, A. Rana, V. Kansal, S. Tanwar, Code Clone Detection – A Systematic Review, in: Proceedings of the Hassanien, A.E., Bhattacharyya, S., Chakrabati, S., Bhattacharya, A., Dutta, S. (eds) Emerging Technologies in Data Mining and Information Security. Advances in Intelligent Systems and Computing, vol 1300. Springer, Singapore, 2021, doi: 10.1007/978-981-33-4367-2_61

[4] Z. Tronicek, Indexing source code and clone detection, Information and Software Technology 144 (2022). doi:10.1016/j.infsof.2021.106805.

[5] J. Martinez-Gil, Source code clone detection using unsupervised similarity measures, in: Proceedings of the International Conference on Software Quality. Cham: Springer Nature Switzerland, 2024. doi: 10.48550/arXiv.2401.09885

[6] V. I. Shynkarenko, I. M. Demidovich, and O. S. Kuropiatnyk, A Dual Approach to Establishing the Authority of Technical Natural Language Texts and Their Components, Science and Transport Progress 2, 102 (2023). doi:10.15802/stp2023/288958.

[7] V. I. Shynkarenko, I. M. Demidovich, Constructive-synthesizing modeling of natural language texts, Computer systems and information technologies 3 (2023). doi:10.31891/csit-2023-3-10.

[8] W. Wang, G. Li, B. Ma, X. Xia and Z. Jin, Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree, in: Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 2020, pp. 261-271. doi:10.1109/SANER48275.2020.9054857

[9] A. Walker, T. Cerny, and E. Song, Open-source tools and benchmarks for code-clone detection: past, present, and future trends. ACM SIGAPP Applied Computing Review 19, 4 (2020). doi:10.1145/3381307.3381310

[10] W. Amme, T. S. Heinze and A. Schäfer, You Look so Different: Finding Structural Clones and Subclones in Java Source Code, in: Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, 2021, pp. 70–80, doi:10.1109/ICSME52107.2021.00013.

[11] I. Zamrii, O. Kulakov Review of automated refactoring approaches for code clones, Connectivity 5 (2024). doi: 10.31673/2412-9070.2024.050575

[12] D. Radadiya, R. Jayakumar, S. Haridass, A Systematic Analysis of Code Clone Refactoring Techniques: A Comparative Review, 2023. doi:10.13140/RG.2.2.33628.87685.

[13] W. Dong, Z. Feng, H. Wei and H. Luo, A Novel Code Stylometry-based Code Clone Detection Strategy, in: Proceedings of the 2020 International Wireless Communications and Mobile Computing, IWCMC, Limassol, Cyprus, 2020, pp. 1516-1521, doi: 10.1109/IWCMC48107.2020.9148302.

[14] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam and B. Maqbool, A Systematic Review on Code Clone Detection, IEEE Access 7 (2019). doi:10.1109/ACCESS.2019.2918202.

[15] Y. Golubev, V. Poletansky, N. Povarov and T. Bryksin, Multi-threshold token-based code clone detection, in: Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 2021, pp. 496-500, doi:10.1109/SANER50967.2021.00053.

[16] W. Dong, Z. Feng, H. Wei and H. Luo, A Novel Code Stylometry-based Code Clone Detection Strategy, in: Proceedings of the 2020 International Wireless Communications and Mobile Computing, IWCMC, Limassol, Cyprus, 2020, pp. 1516-1521, doi: 10.1109/IWCMC48107.2020.9148302.

[17] Y. Hu, D. Zou, J. Peng, Y. Wu, J. Shan and H. Jin, TreeCen: Building tree graph for scalable semantic code clone detection, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. 2022.

[18] D. Yu, Q. Yang, X. Chen, J. Chen and Y. Xu, Graph-based code semantics learning for efficient semantic code clone detection, Information and Software Technology 156 (2023). doi:10.1016/j.infsof.2022.107130.

[19] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo and R. Purandare, Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks, IEEE Transactions on Software Engineering, 48, 10 (2022). doi:10.1109/TSE.2021.3105556.

[20] V. I. Shynkarenko, I. M. Demidovich, Natural Language Texts Authorship Establishing Based on the Sentences Structure, in: Proceedings of the 6th International Conference on Computational Linguistics and Intelligent Systems, COLINS 2022, Volume I: Main Conference, Gliwice, Poland, May 22-23, 2022, pp. 328-337.