Duplicate and Plagiarism Search in Program Code Using Suffix Trees over Compiled Code

Igor Andrianov, Svetlana Rzheutskaya, Alexey Sukonschikov, Dmitry Kochkin, Anatoly Shvetsov, Arseny Sorokin
Vologda State University
Vologda, Russia
igand@mail.ru, rzeyzki@yandex.ru, aas313@mail.ru, kochkindv@bk.ru, smithv@mail.ru, arseny_sorokin@mail.ru

***Abstract*—The search for duplicate source code allow both to improve the quality of the software being developed and to detect plagiarism. In this paper, it is proposed to use a set of features of modern optimizing compilers to simplify and reduce this task to a search by similarity of text fragments. In this case, many types of cosmetic changes in code do not affect the search result. In order to effectively search by similarity, we use sparse suffix trees built on binary encoded data. Algorithms for constructing such a tree and performing a search are presented. The application of the results to detect cheating in a distance programming workshop is described.**

I. Introduction

Duplication of source code occurs naturally in the process of developing and maintaining software. According to estimates in the literature, duplication in industrial software can exceed 50%. Interestingly, for open source products this value is usually many times smaller. Code duplication has a number of negative aspects:

- the growth of both the size of the source texts and binary files,
- decrease in the level of abstraction,
- the need to support multiple copies of almost the same code, etc.

Using tools for searching similarities in the source code allows, in addition to solving these problems, discipline programmers, improve the quality of their program code. It can also be an additional incentive for more effective interaction between developers within the project.

Another important task in this area is plagiarism checking. It is especially relevant for the educational process. The number of training courses in which the development of computer programs is supposed is constantly increasing, as well as the number of students in them.

It's no secret that there are always unscrupulous students trying to pass on someone else's solutions (perhaps slightly modified) or use fragments of someone else's code. To do this, they can change the identifier names, reformat the source text, insert extra variables or unused functions, etc. A simple file comparison does not allow to identify such cases. It requires specialized software.

Taking into account the specifics of the problem being solved, we single out the basic requirements for such software and explain them:

1) Identification of various forms of software code modifications that do not affect the final result. Modifications can range from the simplest (reformatting, renaming identifiers) to quite complex ones (for example, changing language constructions to alternative ones or writing expressions in another form, inserting additional operators and data that do not affect the final result, etc.)

2) The developed software should not require significant computing resources. For example, a module developed by us for controlling plagiarism is used in our university in the system of a remote laboratory workshop on programming. There may be situations when several groups of students work simultaneously with the system. Long delays when checking solutions are highly undesirable.

3) The ability to quickly and easily configure for various programming languages and their dialects. In particular, the aforementioned distance learning system is used when teaching several courses involving various programming languages. The number of such courses is constantly growing, they are dynamically changing. Accordingly, it is desirable that the system has the ability to quickly and easily configure to these changes.

Various approaches are known for solving this problem. So, to identify plagiarism in the program code, it was proposed to use a comparison of such characteristics as the number of operators, operands, special characters, the frequency of references to variables, etc. These approaches are called "attribute counting" [1].

In the case of small modifications of the source text, a simple approach based on the search for common substrings works well. Its extension is parameterized comparison. This is a comparison of texts under the assumption that they can differ only in a systematic change of identifier names. This direction was investigated in the work of B. Baker [2]. To implement such a comparison, a specialized data structure is proposed, which is called the "parameterized suffix tree".

More complex (but revealing more cases of plagiarism) approaches are based on syntactic (and, possibly, partially semantic) analysis of the source texts. An example is the Plague system [3]. It builds the so-called source profiles that reflect the sequence of control structures used in the program.

To compare profiles, one of the varieties of the search for the largest common subsequences is used.

In [4], an algorithm is proposed for detecting duplicate fragments of source code based on call graphs – i.e. oriented graphs representing function calls in programs.

The paper [5] presents a tree-pattern-based method of finding code clones in program files. Duplicate tree-patterns are first collected by anti-unification algorithm and redundancy-free exhaustive comparisons, and then finally clustered.

The work [6] presents the static tracing method in order to improve program plagiarism detection accuracy. The static tracing method statically executes a program at the syntax-level and then extracts predefined keywords according to the order of the executed functions.

There are also other approaches based on comparing syntax trees, graphs of program execution, and some others [7].

Each of the considered methods has one of two drawbacks: it either reveals relatively simple cases of intentional code modifications, or it is strongly tied to a specific programming language and requires difficult adjusting to support another language.

We offer a fairly simple solution that works well in many practical cases. Instead of performing parsing of the code (separately for each language), we can just use the following observations about modern optimizing compilers:

1) The generated code does not depend on the names of local identifiers, text formatting, etc.

2) Different constructions with the same action in many cases are compiled into the same or similar object code.

3) Unused code often does not compile (if the compiler is able to recognize it).

4) When data types change to similar types, the code will not change significantly.

From here, we can conclude that as input data for the search ıt will be effective to use not the source code, but results on compilation (additionally processed in some way).

Acting in this way, the task is reduced to a simpler task of searching by similarity in strings, which can be solved in various ways - for example, with help of suffix trees.

II. USING SOME SPECIFIC COMPILER FEATURES

Now we will consider the above statements in more detail, and illustrate them with examples.

1) The generated code does not depend on the names of functions and local variables, text formatting, etc. This property is clear enough. To use it, it is usually enough to disable writing debugging information to the resulting compilation file.

2) Different syntax constructions with the same action in many cases are compiled into the same or similar object code. To illustrate this statement, we give the following example. Suppose we replace a *for* loop with a *while* loop. Let's look at the result produced by one of the C ++ language compilers (Table I). For convenience, we have enabled the option of generating code in assembler, rather than machine code.

As we can see, the resulting assembler codes are almost identical. The labels have changed slightly, but the sequence of operation codes has remained unchanged. Of course, this does not always work so well. The result may depend on both the compiler and the features of the source text. However, in practice, the results are usually good enough.

It can be concluded that it is quite rational to take intermediate results of compilation as input data when searching for duplicates (it can be assembler text, object code, bytecode, etc.). Then we need to cut out all the extra data, leaving only a sequence of operation codes. Such sequences will be used later to compare programs.

TABLE I. EXAMPLE OF REPLACING OPERATORS WITH ALTERNATIVE ONES

	First file	Second file
Source code	`for(int i=0; i<20; i++) { c+=a; a*=2; }`	`int i=0; while(i<20) { c+=a; a*=2; i++; }`

Compilation result	`xor edx,edx`
	`@3:`
	`add ecx,eax`
	`mov ebx,eax`
	`add ebx,ebx`
	`mov eax,ebx`
	`@5:`
	`inc edx`
	`cmp edx,20`
	`jl short @3`
		`xor edx,edx`
		`@2:`
		`add ecx,eax`
		`mov ebx,eax`
		`add ebx,ebx`
		`mov eax,ebx`
		`inc edx`
		`cmp edx,20`
		`jl short @2`

3) Unused code often does not compile (if the optimizing compiler is able to recognize it). To illustrate this statement, we add an extra variable and operation to the previous example. As we see in Table II, the compilation result has not changed.

TABLE II. EXAMPLE OF INSERTING UNUSED VARIABLES AND OPERATIONS

	First file	Second file
Source code	`for(int i=0; i<20; i++) { c+=a; a*=2; }`	`int i=0; int j=0; while(i<20) { c+=a; j++; a*=2; i++; }`

Compilation result	`xor edx,edx`
	`@3:`
	`add ecx,eax`
	`mov ebx,eax`
	`add ebx,ebx`
	`mov eax,ebx`
	`@5:`
	`inc edx`
	`cmp edx,20`
	`jl short @3`
		`xor edx,edx`
		`@2:`
		`add ecx,eax`
		`mov ebx,eax`
		`add ebx,ebx`
		`mov eax,ebx`
		`inc edx`
		`cmp edx,20`
		`jl short @2`

4). When data types change to similar types, the code will not change significantly. To illustrate it, consider an example of a change of type int (signed integer, 4 bytes) to unsigned int (unsigned integer, 4 bytes).

The changes (Table III) affected only the comparison command: *jb* is now used instead of *jl*. Similarly, when changing the size of the basic types, only registers will change (for example, *eax* to *ax*). Since before comparing we will discard the parameters of the commands, leaving only the operation codes, no additional steps are required to take this into account.

TABLE III. EXAMPLE OF CHANGING OF DATA TYPES TO SIMILAR ONES

	First file	Second file
Source code	`for(int i=0; i<20; i++) { c+=a; a*=2; }`	`for(unsigned int i=0; i<20; i++) { c+=a; a*=2; }`

Compilation result	`xor edx,edx`
	`@3:`
	`add ecx,eax`
	`mov ebx,eax`
	`add ebx,ebx`
	`mov eax,ebx`
	`@5:`
	`inc edx`
	`cmp edx,20`
	`jl short @3`
		`xor edx,edx`
		`@3:`
		`add ecx,eax`
		`mov ebx,eax`
		`add ebx,ebx`
		`mov eax,ebx`
		`@5:`
		`inc edx`
		`cmp edx,20`
		`jb short @3`

The above observations allow us to move from a search for similarities in program code to the next task – to search for similarities of text fragments in the contents of the database. In this case, the sequence of operations codes obtained during compilation will act as input data.

III. PERFORMING A SIMILARITY SEARCH USING SUFFIX TREES

So, we have a database containing many strings that are obtained by compiling and extracting sequences of operation codes. We are faced with the task of efficiently searching in this database records similar to a given one.

Note that the concept of "similarity" can be defined in different ways. In this paper, we use the following method. Two given strings will be considered similar if they have at least k common substrings of length at least l. The constants k and l are selected empirically.

Now we define the problem more formally. Given a set of strings $F=\{F_1,\ldots,F_n\}$, and a string S to search. We need to find a subset of strings $F'\subseteq F$, each of which has at least k common substrings with S of length at least l.

This problem can be effectively solved using generalized sparse suffix trees [8]. A subset of suffixes of input strings is stored in such a tree in a compact way. Each arc is marked with a label – a substring of one of the input strings (Fig. 1.).

To save memory, not the substrings themselves are stored on the arcs, but their positions in the source data: input string identifier, start and end positions of its substring (Fig. 2).

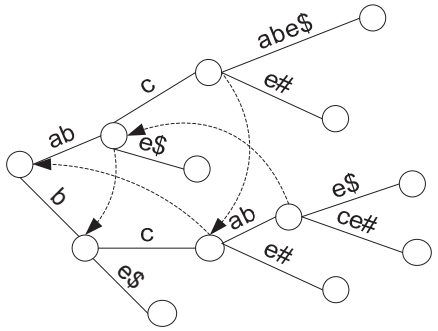


Fig. 1. Example of a generalized sparse suffix tree over strings {“abcabe$”, “bcabce#”}

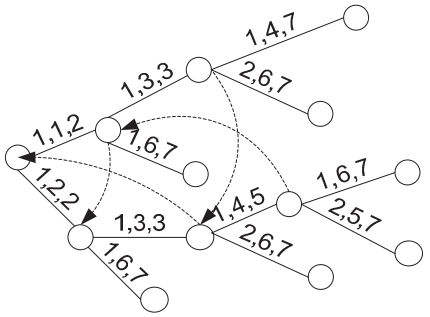


Fig. 2. Compact representation of the tree from Fig. 1 in memory

Concatenation of labels on the path from the root of the tree to each leaf of it uniquely corresponds to one of the suffixes of a particular input line. Each node corresponds to a substring defined as concatenation of labels on the way from the root to this node. The node corresponding to substring *v* will be denoted by <v>.

The dotted lines in Figures 1 and 2 show the suffix links. Suffix links are special arcs that are formed during the operation of the tree construction algorithm. Every suffix link goes from some node <*vw*> to the node <*w*>, where *v* is a nonempty string of minimum length such that the node <*w*> exists in the tree. If such a string does not exist, then the suffix link goes to the root. Suffix links are useful for solving some string processing problems. They are also needed for the ability to add new strings to the tree.

The solution to the above problem is based on the construction of a data structure called matching statistics [9]. Let *t* and *p* are two given strings. Then the matching statistics for *t* with respect to *p* will be the array *ms*[1..|*t*|], where *ms*[*i*] is the length of the largest common substring of t starting at position *i*, which also occurs somewhere in p. Here and below, the notation |*x*| will mean length of *x* (if *x* is a string) or cardinality of *x* (if *x* is a set).

The calculation of the array *ms* is as follows. First, we calculate *ms*[1]. To do this, we need to move from the root of the suffix tree according to the characters of the string *t* until finding a mismatch. Let *t*[1..j] be the matching part of *t*. Then ms[1] = *j*.

To quickly find *ms*[2], the following fact can be used. The

position corresponding to the substring $t[2..j]$ is obviously present in the tree. To get into it, we must follow the suffix link (and after that also perform the so-called canonization – see [1] for more details). Similarly, we can calculate $ms[3]$, and so on.

If the suffix tree for the string p has already been constructed, then the complexity of calculating the matching statistics is $O(|t|)$ or $O(|t|\cdot\log(|\Sigma|))$ depending on the implementation of the tree. Here Σ is the alphabet used. Accordingly, $|\Sigma|$ is the size of the alphabet. In the first case, there is no extra logarithmic factor, but such a tree takes many times more memory and therefore not suitable for long strings.

Let's get back to the original task. Recall its formulation. We are given a set of strings F. It is required to find subset $F' \subseteq F$, each element of which has at least k common substrings with S of length at least l.

To solve this problem, we construct a suffix tree over the string S. This can be done in time $O(|S|)$ or $O(|S|\cdot\log(|\Sigma|))$ using Ukkonen algorithm. After that, we successively calculate the matching statistics for each string $f_i \in F$ and the string S. Knowing the matching statistics, it is easy to calculate the degree of similarity of strings f_i and S.

IV. Effective construction of suffix tree when the input alphabet is large

To efficiently construct a suffix tree, there are several algorithms. One of the most popular is the Ukkonen algorithm. The computational complexity of this algorithm depends on the representation of the tree in memory. If we place in each node a full array of possible transitions of size $|\Sigma|$, then the algorithm works in linear time. However, the tree size in memory will be proportional to $|S|\cdot|\Sigma|$.

For our task, this option is not suitable for the following reasons. Firstly, our alphabet size is quite large. It is equal to the number of different operations in the compiled code and can reach several hundred characters. Secondly, the input files can also be quite large.

To save memory, we can store in each node only those transitions that really come out of it. This greatly saves memory. However, this approach has the following disadvantages.

1) The memory requirements are still quite large. The fact is that we need to create an auxiliary data structure to search for transitions - for example, some kind of balanced tree.

2) This, in turn, complicates the software implementation.

3) Third, computational complexity increases in $\log(|\Sigma|)$ times.

We propose an approach that allows us to eliminate the first two of these shortcomings and significantly reduce the third. The idea is taken from article [10], which shows the possibility of converting the already constructed suffix tree to a binary representation. This is done by replacing all the symbols on the arcs of the tree with their binary codes.

An example of such a suffix tree above the string S="aba$" is shown in Fig. 3. Each arc of the tree is loaded with a substring of the source text, converted to its binary code. To save memory, only positions of substring are stored on arcs. For example, the substring "11000" begins in the 2nd position and ends in the 6th. Detailed explanations for Fig. 3 are given in Table IV.

We will refine this idea as follows. We will not convert the tree to a binary representation after construction. Instead, we will build a tree in this form initially. To do this, we first convert the input string to a binary representation. The converted input string will contain only the characters '0' and '1'.

Next, we need to build a suffix tree over this binary string. However, we do not want all suffixes of this binary string to fall into the tree. Only suffixes corresponding to the positions of the source characters should be included in the tree. In the example, these are positions 0, 8, 16, 24, etc. Such a suffix tree is called an evenly sparse suffix tree.

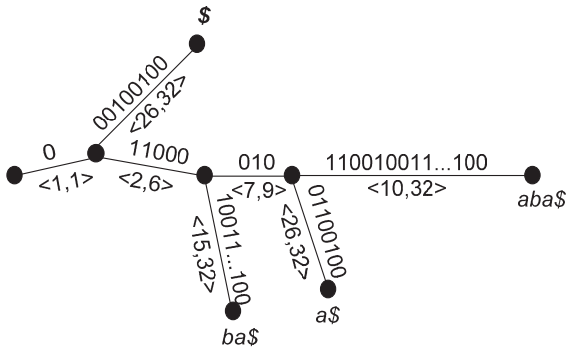


Fig. 3. Binary representation of a suffix tree on "aba$" string

TABLE IV. Explanation of Fig. 3.

Source text	a	b	a	$
Encoded text	01100001	01100010	01100001	00100100
Start positions of suffixes	▲	▲	▲	▲

There is a modification of the Ukkonen algorithm for constructing such a tree in linear time [11]. If the binary representation of one character takes $\log(|\Sigma|)$ bits, then the resulting computational complexity of the algorithm is $O(|S|\cdot\log(|\Sigma|))$.

It is important to note that the number of nodes of such a sparse tree coincides with the number of nodes of a regular suffix tree built over the original string. That is, the size of the tree in memory is at least no larger. In fact, the size is significantly (up to several times) smaller due to the fact that we no longer need an additional data structure to search for transitions from nodes. Indeed, now each node simply stores two pointers: one for '0', and the second for '1'.

In addition, we note that using a binary representation to encode the source text is not the best option. The disadvantage of the binary alphabet is that the resulting tree will also be binary. In this case, the number of internal nodes in such a tree is exactly one less than the number of leaves. But each internal node requires approximately two times more memory than leaf.

By increasing the size of the alphabet into which we code, the proportion of internal nodes decreases in favor of the leaves. This is shown in Fig. 4. However, the time taken to build the tree grows, as can be seen from Fig. 5. Comparing both dependencies, we can conclude that the most suitable size of the alphabet lies in the range from 3 to 5. Given the considerations of the software implementation, it is advisable to take the value 4 as the size of the alphabet, since this is a power of 2. In this case, the processing of each input character is effectively implemented by several processor instructions.

In practice, the proposed approach gives a gain in speed from two times to ten times depending on the input data.

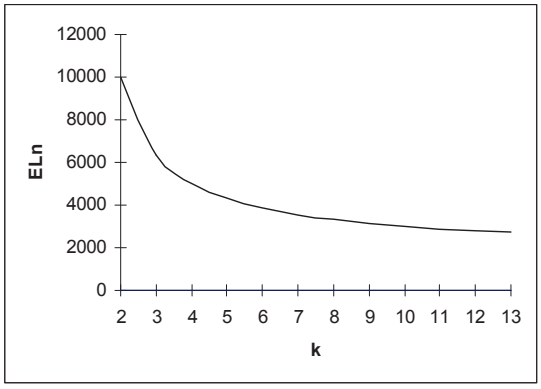


Fig. 4. Typical dependence of the number of internal nodes in the suffix tree *ELn* on the alphabet size *k*

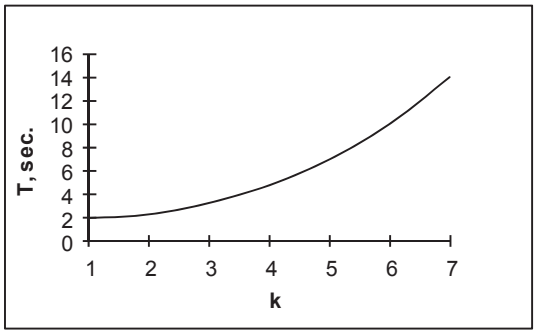


Fig. 5. Typical dependence of the suffix tree construction time on the alphabet size *k*

V. Experimental comparison with other tools

We compared module developed by us with the following two plagiarism detection tools – Jplag and MOSS. Both of these programs are widely known, free to use, and they can be relatively easily integrated into existing e-learning systems. For example, the article [12] describes the experience of developing a plug-in for integrating JPlag and MOSS into Moodle open-source learning environment.

JPlag works as follows. It converts each program into a string of tokens (whitespace, comments, and the names of identifiers are ignored). Additionally, JPlag is able to put some semantic information into tokens. For the comparison of two programs, JPlag then covers one such token string by substrings taken from the other (string tiling) where possible [13].

MOSS uses a similar approach with some differences. For example, it is able to ignore the so-called boilerplate code that is expected in almost all submissions. More details about MOSS can be found in [14].

To perform the comparison, we used the following considerations. Imagine a student who wants to pass someone else's solution into an automatic testing system. Of course, the student does not want plagiarism to be detected. To deceive the anti-plagiarism system, he tries to make some changes to the code not affecting the result of the program. Since the student does not deeply understand the solution, therefore, changes in the code will only be "cosmetic".

To carry out such an experiment, we took the solutions of several quite complex problems (from programming contests) and gave them to several students. We asked them to “cheat” the plagiarism detection systems. Note that the students did not even know the problem statements, they only had the source code.

After analyzing the results, we identified a number of typical ways to intentionally modify the code. Some of them (for example, renaming of variables) were successfully detected by all plagiarism detectors.

However, for several techniques the results were more interesting. These techniques are listed below.

1) Simple replacement of the *for* loop with *while* loop, or vice versa. For instance:

“*while (x < 5) {...}*” changes to “*for(; x < 5 ;)*”,

“*for (int i = 0; i < 5; i++){...}*” changes to

“*int i = 0; while (i < 5){... i++;}*”, etc.

2) Insert a small amount of redundant code. Examples:

x += 0; x++; x--;

3) Insert a small amount of code that never executes (or always executes). Examples:

if (false) {...}

if (true) { a piece of “normal” code from solution }

Table V shows how the results of detecting plagiarism changed on average with the sequential (cumulative) application of the three steps above. In the second and third steps, only about 5% of the extra code was added.

Note: when using programs JPlag and MOSS, we left the default options. With other settings, the results may be somewhat different.

TABLE V. PLAGIARISM DETECTORS COMPARISON RESULTS

Steps used	Similarity level, %		
	JPlag	MOSS	Our module
0 (codes are identical)	100	99 (slightly strange)	100
1	50	54	98
1, 2	28	26	96
1, 2, 3	21	22	96

The results obtained allow us to conclude that the proposed approach works well for a number of typical variants of intentional modifications of source code.

VI. APPLICATION OF THE OBTAINED RESULTS

The results described above were used to implement the plagiarism check module for the distance programming workshop in Vologda State University. This system is available at http://atpp.vstu.edu.ru/acm. Consider some features of this system. A more detailed description can be found in [15].

When checking solutions, the following concept applies. The solution of the problem (user program) is considered as a black box. The program receives input data, performs calculations in accordance with the statement of the problem and saves the results.

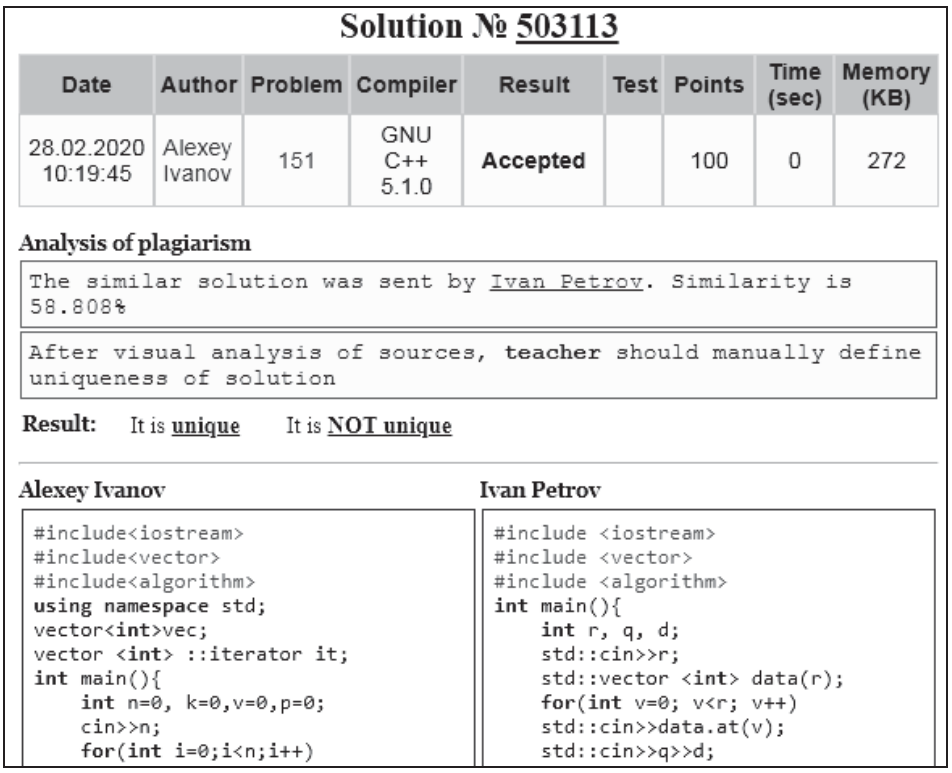
An automated verification system contains a set of tests for each problem. It launches a user solution with various input data and compares the result with the correct test data. In case of an ambiguous answer, a special checking program (so called “checker”) is used. It is interesting that sometimes the complexity of developing a checker exceeds the complexity of solving a problem. Based on the results of the checking, the solution scores a certain number of points.

At the moment, the problems bank includes about two thousand problems of varying complexity. Easy problems are used to teach beginners. More challenging problems are used to prepare students for programming competitions at various levels, including the stages of the World Programming Championship (ICPC). Some local competitions on programming we carry out with the help of this system.

An interesting feature of this system, in comparison with analogues, is the presence of problems not only in "pure" programming. For example, it has a set of assignments for the course “Databases”. In them, students are required to develop SQL queries, as well as stored procedures and functions in PL/SQL language. To check the correctness of the solutions, we developed a special checker. It connects to the Oracle server, executes the code and checks the correctness of the answer.

Another example is a set of problems for the course "Mathematical Logic and Theory of Algorithms". Here, the solution can be, for example, a digital circuit with logic gates, which can be created in the special program and saved as an XML file. A special checker is also used to verify the correctness of such solutions [16].

There is no doubt that the presence of the functionality to identify similar solutions in such a system is highly desirable. Next, we list the features of the plagiarism detection module we developed (its screenshot is shown in Fig. 6).



Solution № 503113

Date	Author	Problem	Compiler	Result	Test	Points	Time (sec)	Memory (KB)
28.02.2020 10:19:45	Alexey Ivanov	151	GNU C++ 5.1.0	**Accepted**		100	0	272

Analysis of plagiarism

The similar solution was sent by Ivan Petrov. Similarity is 58.808%

After visual analysis of sources, **teacher** should manually define uniqueness of solution

Result: It is **unique** It is **NOT unique**

Alexey Ivanov

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
vector<int>vec;
vector <int> ::iterator it;
int main(){
    int n=0, k=0,v=0,p=0;
    cin>>n;
    for(int i=0;i<n;i++)
```

Ivan Petrov

```
#include <iostream>
#include <vector>
#include <algorithm>
int main(){
    int r, q, d;
    std::cin>>r;
    std::vector <int> data(r);
    for(int v=0; v<r; v++)
    std::cin>>data.at(v);
    std::cin>>q>>d;
```

Fig. 6. Example of report about finding similar solutions

1) Plagiarism analysis is performed for any programming language used in the system. Adding support for new programming languages is easy enough. To do this, we need to write a small plug-in for this language, which will perform post-processing of the compiled code to extract a sequence of operation codes.

In many cases, it is convenient to perform a separate compilation process for plagiarism analysis, since we can set specific options for the compiler: for example, generate assembler text instead of binary code.

Note that in the case of interpreted languages, the plugin is often written much more complicated, since we have to process the source code directly. However, many interpreters are able to generate some intermediate code (bytecode, etc.) that can be accessed.

2) The acceptable percentage of similarity can be set for each problem individually. Indeed, for some problems (for example, simple SQL queries), many of the participants' solutions turn out to be very similar to each other. In order for the system not to find many incorrect matches, it is worthwhile to allow a bigger level of similarity. On the contrary, for difficult problems from programming championships, the level of acceptable similarity should be much lower.

3) The system provides the teacher with a list of suspiciously similar solutions. For each item in the list a detailed report is available. An example of such a report is presented in Fig. 6. The report presents both solutions in one window for easy comparison with each other.

After analyzing both solutions, the teacher makes the final decision whether we really have cheating, or this similarity happened by chance. In the first case, the decision is marked as incorrect with the additional status “Not original”.

It is also worth noting that the results obtained with some modifications may be useful for finding duplicate code in ongoing software projects. So, an experimental application for one of the real projects allowed us to identify and eliminate more than 5% of duplicate code.

VI. Conclusion

In this paper, we have proposed a simple and easily implemented approach to finding duplicates and plagiarism in program code. Using post-processed compilation results as input to the search allows you to automatically ignore many cosmetic changes in the code that do not affect the result of its work. As a result, the task is simplified and reduced to determining the presence of common fragments in text data.

To solve the obtained problem, suffix trees were used. A feature of our task is the relatively large size of the alphabet, which negatively affects the performance of this data structure and also its size in memory. We solved this problem as follows.

Based on the ideas proposed in [10], we developed an algorithm for constructing the binary suffix tree by constructing a uniformly sparse suffix tree over pre-encoded text. The resulting gain is highly dependent on the input data, it varies on average from two to ten times.

The results were successfully used in the plagiarism control module for a distance programming workshop at Vologda State University.

It should be noted that the proposed method for constructing sparse suffix trees over pre-encoded text is quite universal and suitable not only for the considered problem, but can also be successfully used for many other applications.

References

[1] S. Grier, “Tool that Detects Plagiarism in PASCAL Programs”, SIGSCE Bulletin, Vol. 13, 1981.

[2] B. Baker, ”A theory of parameterized pattern matching: algorithms and applications”, in Proc. of the 25th ACM Symp. on the Theory of Computing, 1993, pp. 71-80.

[3] Paul Clough, “Plagiarism in natural and programming languages: an overview of current tools and technologies”, Department of Computer Science, University of Sheffield, 2000. Web: http://www.dcs.shef.ac.uk/~cloughie/papers/Plagiarism.pdf

[4] T. Cholakov, D. Birov, “Duplicate code detection algorithm”, CompSysTech '15: Proc. of the 16th Int. Conf. on Computer Systems and Technologies, June 2015, pp. 104–111.

[5] Hyo-Sub Lee, Kyung-Goo Doh, “Tree-pattern-based duplicate code detection”, DSMM '09: Proc. of the ACM first international workshop on Data-intensive software management and mining, November, 2009, pp. 7–12.

[6] Jeong-Hoon Ji, Gyun Woo, Hwan-Gue Cho, “A source code linearization technique for detecting plagiarized programs”, ITiCSE '07: Proc. of the 12th annual SIGCSE conference on Innovation and technology in computer science education”, June 2007, pp. 73–77.

[7] I. Andrianov, A. Grigorieva, “Effective search for plagiarism in program code for a remote programming workshop system”, in Proc. of the international scientific and practical conference "Informatization of engineering education (INFORINO-2016)”, Moscow: MEI Publishing House, 2016, pp. 485–488.

[8] S.F. Svinyin, I.A. Andrianov, “Application of evenly sparse suffix tree for string processing tasks”, SPIIRAS Proceedings, No. 3 (34), 2014. pp. 247-260.

[9] Dan Gusfield, “Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology”, Cambridge University Press, 1997, 534 p.

[10] A. Anderson, S. Nilsson, “Efficient implementations of suffix trees”, Software – Practice and Experience, 1995, Vol. 25, pp. 129-141.

[11] Juha Kärkkäinen, “Sparse suffix trees”, Proc. of the 2nd Annual International Conf. on Comput. and Combinatorics. Lecture Notes in Computer Science, 1996, Vol. 1090, pp. 219-230.

[12] Thanh Tri Le Nguyen, Angela Carbone, Judy Sheard, Margot Schuhmacher, "Integrating source code plagiarism into a virtual learning environment: benefits for students and staff", ACE '13: Proceedings of the Fifteenth Australasian Computing Education Conference, Volume 136, January 2013, pp. 155–164

[13] Lutz Prechelt, Guido Malpohl, Michael Philippsen, “Finding Plagiarisms among a Set of Programs with JPlag”, Journal of Universal Computer Science, vol. 8, no. 11 (2002), pp. 1016-1038.

[14] Saul Schleimer, Daniel S. Wilkerson, Alex Aiken, “Winnowing: Local Algorithms for Document Fingerprinting”, SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, June 2003, pp. 76–85.

[15] I.A. Andrianov, N.O. Menuhova, “Refinement of a remote workshop of Vologda SU for checking problems on informatics according to the rules of school olympiads”, Proc. of Int. scientific and practical conference “Modern society, education and science”, Tambov, 2014, pp. 10-13.

[16] I.A. Andrianov, “Automation of checking of logic circuits”, Proc. of tenth international scientific and technical conference “Intellectual Information Technologies and Intellectual Business (INFOS-2019)”, Vologda: Vologda State University, 2019, pp. 114-117.