UDC: 004.9:004.43

ALGORITHM AND SOFTWARE IMPLEMENTATION OF MODELS FOR PROGRAM CODE COMPARISON ON THE EXAMPLE OF C-LIKE PROGRAMMING LANGUAGES

Koshovyi R. V.[1], Kirei K. O.[2]

[1,2] Petro Mohyla Black Sea National University, Mykolaiv, Ukraine
ORCID: https://orcid.org/0009-0005-2951-6574, https://orcid.org/0000-0002-9338-2380
E-mail: romanrokel@gmail.com, kirey.kea@gmail.com

DOI: https://doi.org/10.15673/atbp.v15i4.2722

Abstract. *This scientific work introduces an innovative algorithm and its software implementation for addressing the issue of detecting code similarities and identifying instances of code borrowings in software solutions. This development, pre sented in both finished and abstract forms, has the potential to revolutionize the way educators and institutions approach evaluating the creative solutions of student-programmers. The algorithm is designed to automate checks on software solutions submitted by students for various programming challenges, competitions, and Olympiads. The tool provides an effective means of detecting code similarities and flagging potential instances of plagiarism, allowing instructors to ensure that the submissions are original and the result of conscientious efforts. The work presents two examples of creative tasks that require careful evaluation for code similarity, providing concrete illustrations of how the algorithm works in practice. The algorithm itself is described in detail, and parts of the software code implementation in C# programming language are provided, enabling interested parties to explore the tool's technical workings. The results of testing and analysis of the algorithm are presented, with the work highlighting the potential of the tool to automate time-consuming and resource-intensive evaluations of software solutions. Visualizations of the test results provide clear and concise insights into the efficacy of the algorithm, enabling educators to assess the originality of the solutions with ease. Overall, the work is an invaluable contribution to the field of software development, providing educators with an innovative tool for ensuring the originality of software solutions submitted by students. With the growing need for transparency and academic integrity in higher education, this development has significant potential to transform the way institutions approach software solution evaluations, empowering students to pursue creative problem-solving with confidence. All of this can be achived by this algorithm being implemented in the future in bigger application as it`s main or side feature.*

Анотація*. У цій науковій роботі представлено інноваційний алгоритм та його програмну реалізацію для вирішення проблеми виявлення подібності коду та виявлення випадків запозичення коду в програмних рішеннях. Ця розробка, представлена як у закінченій, так і в абстрактній формах, має потенціал революціонізувати підхід викладачів та установ до оцінювання творчих рішень студентів-програмістів. Алгоритм призначений для автоматизації перевірки програмних рішень, поданих учнями на різноманітні конкурси, конкурси та олімпіади з програмування. Інструмент забезпечує ефективний засіб виявлення схожості коду та позначення потенційних випадків плагіату, дозволяючи викладачам переконатися, що подані матеріали є оригінальними та результатом сумлінних зусиль. У роботі представлено два приклади творчих завдань, які потребують ретельної оцінки подібності коду, надаючи конкретні ілюстрації того, як алгоритм працює на практиці. Детально описано сам алгоритм, а також надано частини реалізації програмного коду мовою програмування C#, що дозволяє зацікавленим сторонам досліджувати технічну роботу інструменту. Наведено результати тестування та аналізу алгоритму, а також підкреслено потенціал інструменту для автоматизації трудомістких і ресурсомістких оцінок програмних рішень. Візуалізація результатів тестування дає чітке та стисле уявлення про ефективність алгоритму, що дозволяє викладачам легко оцінити оригінальність рішень. Загалом робота є неоціненним внеском у сферу розробки програмного забезпечення, надаючи педагогам інноваційний інструмент для забезпечення оригінальності програмних рішень, поданих студентами. Зважаючи на зростаючу потребу в прозорості та академічній доброчесності у вищій освіті, ця розробка має значний потенціал для зміни підходу установ до оцінювання програмних рішень, надаючи студентам можливість впевнено творчо вирішувати проблеми. Усього цього можна досягти шляхом реалізації цього алгоритму в майбутньому у більшій програмі як основної чи додаткової функції.*

Keywords: Algorithm, code comparison, similarities, program solution, program implementation, analysis automation
Ключові слова: алгоритм, порівняння коду, подібності, програмне рішення, програмна реалізація, автоматизація аналізу

I. INTRODUCTION

To support the technological process and realize its capabilities, as well as its availability to the masses, the IT industry must expand and improve, which obviously requires more specialists. In addition, the development of software and the latest devices constantly requires innovative solutions from specialists. That is why potential IT specialists must not only be able to write high-quality code, but also find quick solutions to non-trivial problems.

On the way to becoming a highly qualified specialist, an important role is played by higher education institutions. In addition to teaching students about technology and the ability to write high-quality code, they are often also presented with atypical problems that do not have a "linear" solution, but require creativity from the future specialist. Accordingly, each solution will also be atypical and different from the others.

It is important that the verification of the work performed by the student involves not only determining the correctness of the given solution to the problem but also the absence of dishonest execution of the task, for example, by borrowing the code. Tasks that require the student to realize his creative potential often do not have a clear solution, so it is especially important to check for a "copied" solution.

However, students create hundreds and thousands of lines of code every day, and checking their uniqueness manually is unreasonable in terms of time and effort. Online services for checking texts for uniqueness do not take into account the peculiarities of files with program code, so a special tool is needed to perform such checks.

Such a tool, however, will not execute the code, or compile/interpret it - due to many factors and the specifics of the task, this is not a major and important but it will take a very long time to process and execute. Instead, what is needed is a solution that, based on the analysis of the program text, taking into account the features of the programming language in which the code is written, will perform a comparison and conclude whether it is really the author's solution or whether the code has been copied in bad faith.

II. LITERATURE ANALYSIS

Before solving the problem, it is important to consider already existing applications that solve this or a similar problem (Dupli Checker [1], Plagiarism Checker [2], W3docs [3], Diffchecker [4]), after which - pay attention to the target tasks, the solution code of which will be checked by the given algorithm.

2.1. Analogues

The Internet is full of various (usually conditionally free) applications and services that provide an opportunity to compare texts and/or files. In this case, it is important to consider 2 types:

- text checking services for plagiarism
- services for comparing two files for difference

The first type is aimed at plain text - it checks for the presence of its individual elements in the Internet, after which it provides a result in the form of a percentage ratio of the total size of the text borrowed from elsewhere to the size of the original text. The problem is that such services do not take into account the peculiarities of files with program code - the methods used by checkers to check the program code are simply not relevant. Examples of such online services are Dupli Checker and Plagiarism Checker.

Another type, which finds the difference between files, is partially adapted to the specifics of code files so that it can work fine with them, but has a different purpose. Such services do not find borrowings, but perform a sequential comparison, which can be useful when tracking the changes made. This will show the change in the string exactly to the character. Similar applications are built into IDEs as a way to find out, for example, what changed in the code in a commit. However, with such services, it is sometimes difficult to assess whether two files with program code have any relation to each other, for example, as an original and a slightly modified copy of a solution (Fig. 1). That`s services like W3docs and Diffchecker.

To obtain an application that will analyze 2 files and find out whether one of them is a copy or a modification of the other, and to what extent, already existing applications of the types described above have been developed. Namely, the conclusion is drawn about the need to combine the search for coincidences with consideration of the specifics of the files to be checked, i.e., the syntax of the programming language in which they are written.

Fig. 1 – comparing 2 versions of 1 file using Diffchecker

2.2. University task example

As it was said earlier, tasks whose solution code will need to be checked according to the algorithm provided in this work can be a creative work for students at a university, or a task from a student programming contest. The developed application was tested on solutions for both types of tasks, for which, in fact, the algorithm was developed.

An example of task for creativity is the task from [5, P.40-41]:

Write a program using the C/C++ language to analyze the contents of the mails.json file (mails.zip archive) and get answers to the questions given after the task description. Form the results in the form of an essay.

Correspondence of Enron employees (file mails.zip). This file contains the correspondence of approximately 150 users, mostly Enron executives. Since the complete dataset takes up approximately 1.5 GB when unpacked, the body field has been removed from it to reduce the size.

The file contains lines of something like this:

{ "_Id": ObjectId ("4f16fc97d1e2d32371003e27"),
"SubFolder": "notes_inbox",
"Mailbox": "bass-e",
"Filename": "450.",
"Headers": {
"X-Folder": "\\Eric_Bass_Dec2000\\Notes Folders\\ Notes inbox",
"X-bcc": "",
"To": "eric.bass@enron.com",
"X-From": "Michael Simmons",
"Message-ID": "<6884142.1075854677416.JavaMail.evans@thyme>",
"Mime-Version": "1.0",
"From": "michael.simmons@enron.com",
"X-To": "Eric Bass",
"X-cc": "",
"Content-Transfer-Encoding": "7bit",
"X-Origin": "Bass-E",
"Subject": "Re: Plays and other information",
"X-FileName": "ebass.nsf",
"Date": "Tue, 14 Nov 2000 8:22:00 -0800 (PST)",
"Content-Type": "text/plain; charset=us-ascii" }
}

Row:

{ "_id" : { "$oid" : "4f16fc97d1e2d32371003e27" }, "subFolder" : "notes_inbox", "mailbox" : "bass-e", "filename" : "450.", "headers" : { "X -cc" : "", "From" : "michael.simmons@enron.com", "Subject" : "Re: Plays and other information", "X-Folder" : "\\Eric_Bass_Dec2000\\Notes Folders\\ Notes inbox", "Content-Transfer-Encoding" : "7bit", "X-bcc" : "", "To" : "eric.bass@enron.com", "X-Origin" : "Bass-E" , "X-FileName" : "ebass.nsf", "X-From" : "Michael Simmons", "Date" : "Tue, 14 Nov 2000 08:22:00 -0800 (PST)", "X-To " : "Eric Bass", "Message-ID" : "<6884142.1075854677416.JavaMail.evans@thyme>", "Content-Type" : "text/plain; charset=us-ascii", "Mime-Version" : " 1.0" } }

1. Determine the total number of letters ("_id" key).

2. Determine the number of unique senders ("From" key).

3. For each day of the week, count how many emails were sent to ebass@enron.com (key "To")?

4. Analyze the correspondence between Shanna Husser and Eric Bass. How many letters did each of them send to the other?

5. Laurie Ellis sometimes sends letters with the same subject ("Subject" key). For each email subject, count how many times it was used in the year 2000.

6. On which day of the week was the maximum number of letters sent?

In short, it is necessary to automate the verification of correspondence data stored in json format in order to obtain from them the information required by the user. The task is expected to be performed in the C/C++ programming language. To solve this problem, it is necessary to develop an algorithm that will be able to separate the data required for a certain task from others, and correctly interpret the results to the user. The solutions can be quite diverse and use different classes, tools, functions and facilities available in one or another programming language. Also, each of the 6 tasks requires the program to focus on different information from the given data, which prompts either the writing of large code with the possibility of choosing an action, or different, remotely similar applications specified for individual tasks. The application from this work is focused on such tasks.

2.3. Task from programming contest

Sports programming is somewhat similar - contestants have a certain number of tasks. In each such problem, there is a condition that describes the history, the input data available to the program, and what the program needs to calculate based on this data. Typically, sports programming also provides one or more sets of input and output data as an example for familiarization and initial testing of solutions. Many Ukrainian participants of programming contests use site eolymp.com [6] for preparation, that has amount of task from different contests and courses big enough to train and learn. For example, here is one of the tasks available there:

The vouchers

The tour operator did not sell n (n < 15) vouchers to mountain-ski resort because of the great frost. The terms of vouchers validity is already come. To reduce the losses, it was decided from the February 1, that all the vouchers for which remained dk (dk ≤ 30) days, to sell for minimal cost – for ck (ck ≤ 100) UAH for a day only for that days, which remained from the sale day (k = 1..n).

What is the largest amount of income can get the travel company selling these vouchers, if one voucher can be sold only in one day?

Input

The first line contains the number of vouchers n. Each of the next n lines contains two numbers - the number of days dk left and the cost of each day ck.

Output

The maximum amount of income.

Input example #10

```
4
2 37
3 45
1 46
4 30
```

Output example #10

```
232
```

Within the framework of one contest, where a limited number of teams take part and have a limited amount of time (for example, for ICPC it is 5 hours), it is quite likely that among the participants there will be teams with the same view and method of solving a certain problem, but it is unlikely (rather, even, almost impossible) that the program code of their solutions will match exactly if they solved the problem in good faith. If the code still matches, this calls into question both solutions and will require a more detailed review of the situation by the judges.

2.4. Conclusion

It is worth paying attention to the fact that in both tasks the solution is expected in the form of a program in one programming language and in one file.

To correctly process the program code of a certain programming language, you need to be familiar with at least its basic features and syntax. In this work, the proposed algorithm will also be implemented for C-like programming languages C, C++, C# and Java, which requires familiarization with the relevant documentation and literature [7-15]. The choice of programming languages is justified by the fact that this set of languages is currently a fairly common starting

point for learning programming in principle, and is taught to students at universities. Also, C++ is one of the most popular languages for solving sports programming problems due to its speed of execution.

The abstraction of the algorithm presented in this work is not tied to a specific programming language - adaptation to specifics is important precisely for the implementation of this algorithm.

III. OBJECT, SUBJECT, AND METHODS OF RESEARCH

Object of research is methods of identifying codes for similarity.

Subject is a method of analyzing program codes in example of C-like languages for similarity by determining the fundamental points in checking the codes for similarity and highlighting the key features of the syntax.

Research methods - analysis of theoretical information, analysis of methods of checking codes for similarity, analysis of the syntax of target programming languages, testing of the developed algorithm.

3.1. Code replacement

The initial version of the program is designed to check files with the code of the most popular programming languages, namely C, C++, C#, Java. Given the peculiarities of these languages, when trying to borrow code, novice programmers can resort to various modifications of the code, which will not affect the progress and results of the program, but will change its general appearance in such a way that the fact of borrowing cannot be seen with the naked eye. To implement a file comparison application with software code, you need to take into account the most likely code changes.

Renaming is the easiest way to change the appearance of code (Fig. 2). For most IDEs, this operation is performed in a few clicks. However, regardless of whether the name of a variable, class, method, constant, or any other code element has changed, the result of the program execution will not change. In order to recognize such substitutions, a deep analysis of the code is required at the level of parsing it down to syntactic units, where the name can already be distinguished from the operator or keyword. At the same time, the check by ordinary character-by-character comparison will be incorrect only in the case of replacing one name with another, provided that the lengths of their names are different – it is easier to bypass this feature.

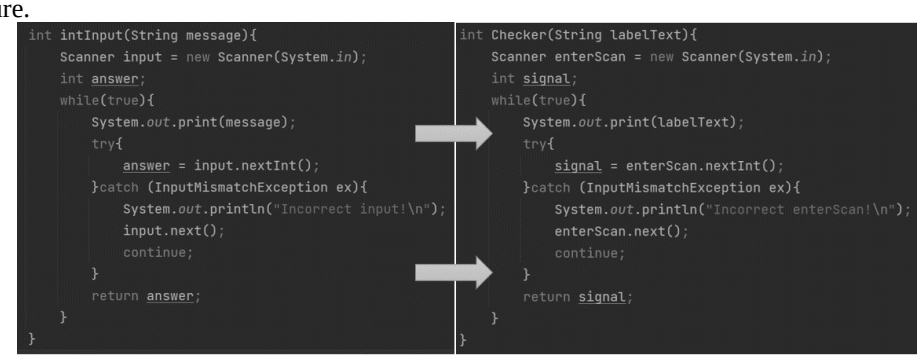

Fig. 2 – naming replacement

Changing the order of commands, methods, classes, and other code elements is a more interesting process, because sometimes with this technique, the code can be turned upside down, and at the same time it will remain executable, but unrecognizable (Fig. 3). Nevertheless, there are many cases when lines of code simply cannot be moved to another place, because they perform their function only there, where they are - this prevents the free abuse of such a technique. A program for detecting the coincidence of two code files should eliminate the effect of any "scrambles".

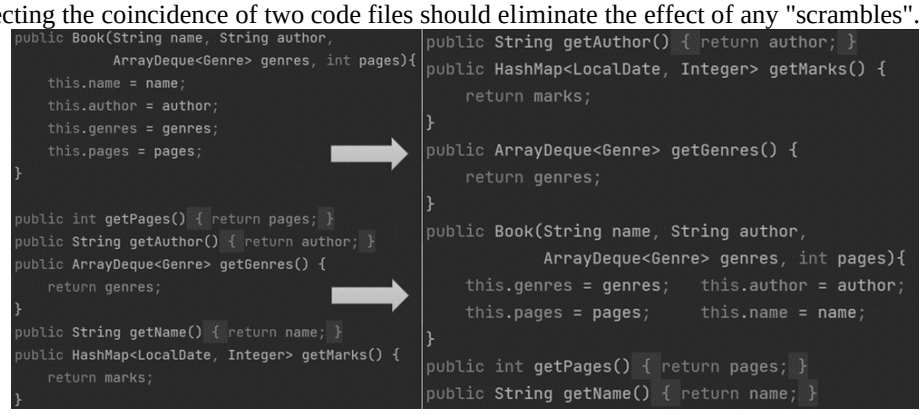

Fig. 3 – changing the command order

Finally, it is worth considering the presence in the code of preprocessor commands, linking libraries, namespaces or their elements, comments, and string literals (Fig. 4). Typically, they will either be the same for most software applications that perform the same task, or their presence or content will not affect the execution of the program or its results at all.

They can also be specially changed and instructed in the code (or removed from it) in order to change the general appearance - then it will be easy to visually confuse the files (although the de facto program code has hardly changed). Taking into account these elements when checking for convergence of program codes will reduce accuracy of the result, so it was decided to ignore them when scanning.

```cpp
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <cstring>
#include <windows.h>
#include <cstdlib>
#include <string>
#include <conio.h>

using std::string;
using std::ifstream;
using std::endl;
using std::cout;
using std::cin;
```

a)

```java
package com.company;

import org.junit.jupiter.api.*;
import org.junit.jupiter.api.condition.EnabledIf;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import org.junit.jupiter.params.provider.ValueSource;

import java.time.Duration;
import java.time.LocalDate;
import java.util.*;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
```

b)

```cpp
int max_base_coeff_value = 7; //max coefficient operator can have itself
char last_symbol = NULL;
bool previous_oper = true; //is previous symbol of string operator or not
/*Errors block and value to save the error-condition
of current session. Will be used for input validation and to prevent ecxeptions*/
bool error_flag = false; //is error at all or not
map <string, bool> error{ //array for all errors
    {"Dot error! Too many dots in number!", false},
    {"Uncorrect symbols or unknown operation!", false},
    {"Uncorrect problem`s end!", false},
    {"Negative factorial error! Result can be incorrect!", false},
    {"Subfactorial expression isn`t natural! The answer may be uncorrect!", false},
    {"Uncorrect operators usage!", false} };
```

d)

Fig. 4 – code elements that are not important to check: a – connecting libraries, namespace elements, preprocesses in C++, b – in Java, c – the use of comments and string literals in C++

3.2. Algorithm

When mixing all the methods of changing the program code given in the previous paragraph, the final result will be completely unrecognizable compared to the original file, so the program for determining the level of convergence should take into account all the ways of hiding code borrowing, including their use together randomly.

Among the possible options for solving the problem, it was chosen to decompose the program code into parts so as to obtain a tree-like structure (in other words, a hierarchy), and to compare trees by their leaves and branches.

In more detail, the program code will be decomposed into small elements and arranged in the form of a tree with four levels:

first level (the lowest, leaves) is a syntactic unit (operator, variable or constant, method, keyword, etc.) (hereinafter referred to as a “word”);

second level - command;

third level – block of code;

fourth level (the highest, the root) - all the program code.

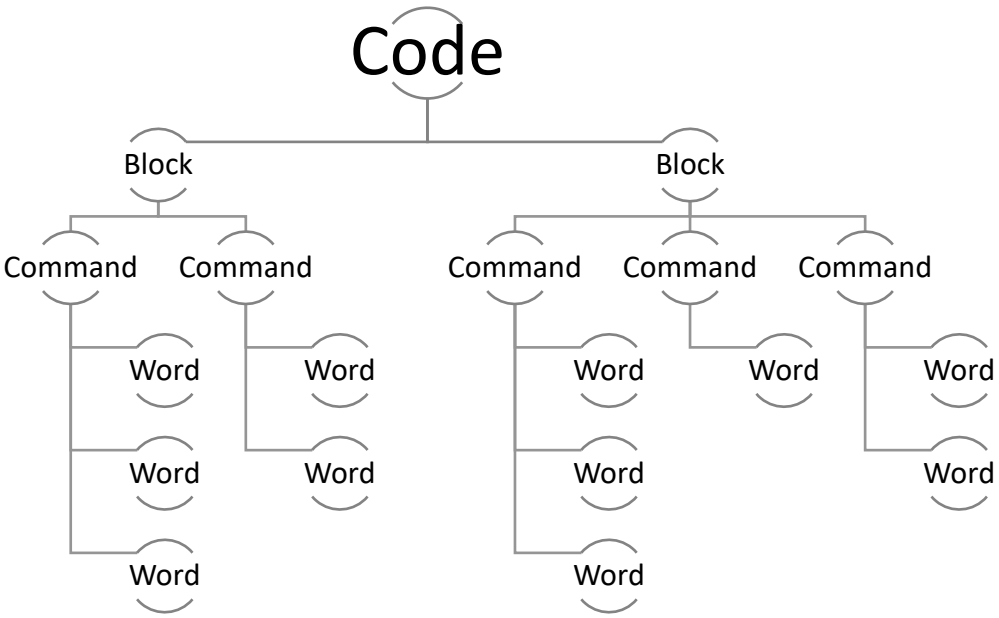

Fig. 5 – schematic representation of the decomposed code tree

This structure makes it possible to compare the relevant elements with each other by their nature and role for the code, instead of searching for every possible element throughout the code. Compared to the search "everywhere", this is an accelerated solution to the problem of finding code similarities in the event that an unscrupulous programmer uses permutations of commands (lines) and blocks of code, because in a hierarchical structure the order of the element loses its meaning. When creating code trees, it is important to exclude the possibility of getting there preprocessor commands, connecting libraries, namespaces or their elements, comments, string literals, as well as spaces, tabs, transitions to a new line, so that the comparison takes place only on significant data of files with program code.

Next, each element of a certain level will be compared with the corresponding elements of another code and store as a result the highest level of similarity found. For higher levels, a lower-level comparison will be called.

Given the above, the program will consist of two main parts - a decomposer (which decomposes the program code from a file and forms a code tree), and a comparator (actually, a part that will compare 2 code trees created by the decomposer). Since the execution time can reach tens of seconds with large amounts of software code, it is also necessary to constantly notify the user about the progress of the work, and at the end a method will be called that will return a clear result that will not change from check to check, provided that the files are unchanged, which are checked.

3.3 Decomposer

For the correct decomposition of the code and selection of only the necessary elements, it is important to clearly define the features of the programming languages under consideration, namely, C, C++, C#, Java.

First, this is the beginning of the code, which probably or one hundred percent contains:

– #define, #include in C, C++;

– using, namespace in C, C++, C#;

– package, import in Java.

Secondly, for all the programming languages under consideration, the following symbols have been identified that will allow you to control the place of an element in the code tree:

– “whitespace”, square brackets, round brackets, period or mathematical operator for "word";

– “;” (semicolon) for command;

– “{ }” (curly brackets) for a block of code;

Finally, the most popular C-like programming languages have the same notation for comments and string literals:

– “//” (double slash) for a one-line comment;

– “/* … */” (slash-star, comment, star-slash) for a multi-line comment;

– “" … "” (double quotes, text, double quotes) for a string literal;

– “'… '” (single quote, text, single quote) for character [7, 10-12, 14].

Since the tree must be free of spaces, tabs, and other similar characters (defined in List<chars> spaces), a function was developed, the call of which will accompany all program code to skip non-significant characters:

```
int SkipSpace(string str, int pointer)
{
	while (pointer < str.Length && spaces.Contains(str.ElementAt(pointer)))
	{
		pointer++;
	}
	return pointer;
}
```

After the decomposer is called, it scans the text given to it, and for all function calls (such as SkipSpace()) it passes the position where it stopped, and the function returns the position from which to continue.

In the implementation, the position is called a pointer because of its similarity to the usual pointer in programming - it goes through the entire code from beginning to end according to the decomposer algorithm, which already analyzes every symbol that falls under the pointer.

A nested List<> collection was used to preserve the hierarchy of elements of the decomposed code.

First of all, the DecomposeCode() method skips a group of preprocessor commands, linking libraries, etc., because they are usually placed at the beginning of the program code. Together with the SkipSpace() method, the loop will not terminate until a command not listed in the parameters is encountered in the code being tested, where the parameters list given as follows:

```
readonly List<string> precodeDirectives = new() { "#include", "#define", "using", "import", "package", "namespace" };
```

Decomposer performs character-by-character scanning of the code provided to it starting from the position given to it in the parameters and has its own procedure for each character from the "special" ones discussed earlier. So, if a slash is detected, the next character is immediately checked and a conclusion is made as to whether it is the beginning of a comment, and if so, the end of the corresponding comment is searched using String.IndexOf(). This is also the case with string literals, but when searching for the end of a string literal, the possibility of escaped quotes is taken into account:

```
if (code.ElementAt(pointer) == "") //text appearance
{
	do
	{
```

```
                pointer = code.IndexOf("", pointer + 1);
            }
            while (pointer < code.Length && code.ElementAt(pointer - 1) == '\\');
            pointer = SkipSpace(code, pointer + 1) - 1;
            continue;
        }
```

Next is the symbols, appearance of which can signal of the fact current word that is being processed, is finished (can be modified same as precodeDirectives):

```
readonly List<char> wordEnds = new() { '\n', '\r', '\t', ' ', '(', ')', '[', ']', '+', '-', '/', '*', '.' };
```

If the current symbol is not the beginning of a comment or a string literal, the decomposer modifies its existing branch depending on this symbol. If the character is a word end character (defined in List<chars> wordEnds), then the word is saved and added to the current command branch. If the decomposer encounters a ";" (end-of-command), then the last word in the command branch is stored in the block branch.

Decomposer is recursive – when the curly brace opening character "{" is detected, the Decomposer method calls itself and processes what is inside the curly braces as a separate block of code. When a closing curly brace character is detected, the decomposer saves the last word, the command, and adds the entire block that came out to the common tree, after which it returns to processing the block of code that was processed by the recursion step. Because of this, in the code tree, all blocks are equal, regardless of their nesting level. Also, the phrase "My vassal's vassal is not my vassal" is now appropriate for code blocks, i.e., if there is a code block #1 and there is a code block #2 in it, when expanded and added to the code tree, the expanded code block #1 will not have code from block #2, since block #2 will have no connection with the first block.

3.4 Tree Comparator

The comparator is a group of methods - the CompareCodes() code comparator method, the CompareBlocks() block comparator method, and the CompareCommands() command comparator method. First, the first method is called for the two codes, which, before scanning, uses the CalculatePointRange() method to set the maximum possible threshold of points that can be obtained.

Here, scores are a method of measuring similarity. After analyzing the code tree, the CalculatePointRange() method determines the maximum possible number of points that can be obtained as a result of checking two program codes for similarity. In other words, this is the result that would be given if the two code trees were completely identical. The calculation of the percentage similarity value will be based on this value. For each coinciding symbol in the corresponding team, 1 point will be awarded, and for a completely identical word – 5:

```
range += word.Length + 5; //for each word
```

Next, a points counter is created and scanning begins. To a code comparator, it looks like a mapping of all code blocks from the first tree to all code blocks from the second tree. For each block from the first tree, the score obtained when comparing it with the most similar block from the second tree is stored, that is, the best score is stored. The results for each block are accumulated in the counter.

The block comparator works similarly - it compares each row from the first block with each row from the second, and saves the best results (from the most similar rows), the sum of which is returned to the block comparator. Finally, the command comparator already does the main part of the work.

Two tree branches that contain words fall into the command comparator. Here they go through 3 stages of scanning. During the first stage, matching words are compared, and for each match, 5 points are added to the counter.

Next, for each of the branches that are compared, a line is formed that contains all "leaves" (words) without separators. After that, the resulting strings are compared character by character like the words in the first stage, first in forward order (2nd stage) and then in reverse (3rd stage). The results of these "rough" comparisons are compared and the number of positions that "matched" is added to the counter, that is, the symbols in them were the same in one or two scans at once, and then the number in the score counter is returned to the block comparator method.

The entire comparator is built in such a way that for the code laid out in the form of a tree, it searches for the most similar elements to each corresponding element in the other code. In other words, the comparator looks for bits of code #1 in code #2. Except in the case where the same trees were built from the code in the two files being checked, the results of searching for elements of code #1 in code #2 and searching for elements of code #2 in code #1 will be different.

Suppose that code #1 is completely borrowed from code #2, but is not a copy of it, but a part. In this case, if we search for elements from the first tree in the second, the result will be 100% - every line or block of code that we can find in the first code file exists in the second file. However, if you check the other way around, the result will not be unambiguous - depending on which part of code #2 makes up code #1, it will vary. When the scan is run, there is no information about potential borrowing in one file with the program code from another.

Due to this feature, in this program, the code comparator is called twice, and their results are compared - the final result of the check is considered to be the larger of the two results:

```
Parallel.Invoke(
    () => CompareCodes(tree1!, tree2!),
    () => CompareCodes(tree2!, tree1!)
);
```

In the decomposition and comparison, 2 similar tasks are performed - the capabilities of the C# programming language make it quite easy to parallelize the execution of these tasks (using Parallel.Invoke()), which for a multi-core processor (currently - almost every one) speeds up the execution of the algorithm twice!

That`s all for the implementation of the algorithm. It is enough to put the implementation in a ready-made application or create one to visualize the data that can be obtained. To demonstrate these, a minimalistic UI was created for this work.

IV. RESULTS

As a result of the implementation of the algorithm, we have several classes that perform the tasks of decomposition, comparison and combination of results. The application for this work was developed using ASP.NET MVC - here, when creating a response to the user, the controller reads the provided files or code and passes the code in the form of a class string, which already organizes the comparison using the implemented algorithm classes and returns comparison results in specified object. The controller then returns the result of the comparison and some information about the process of the algorithm to the user (e.g., execution time). The use of the application is depicted in Fig. 5.

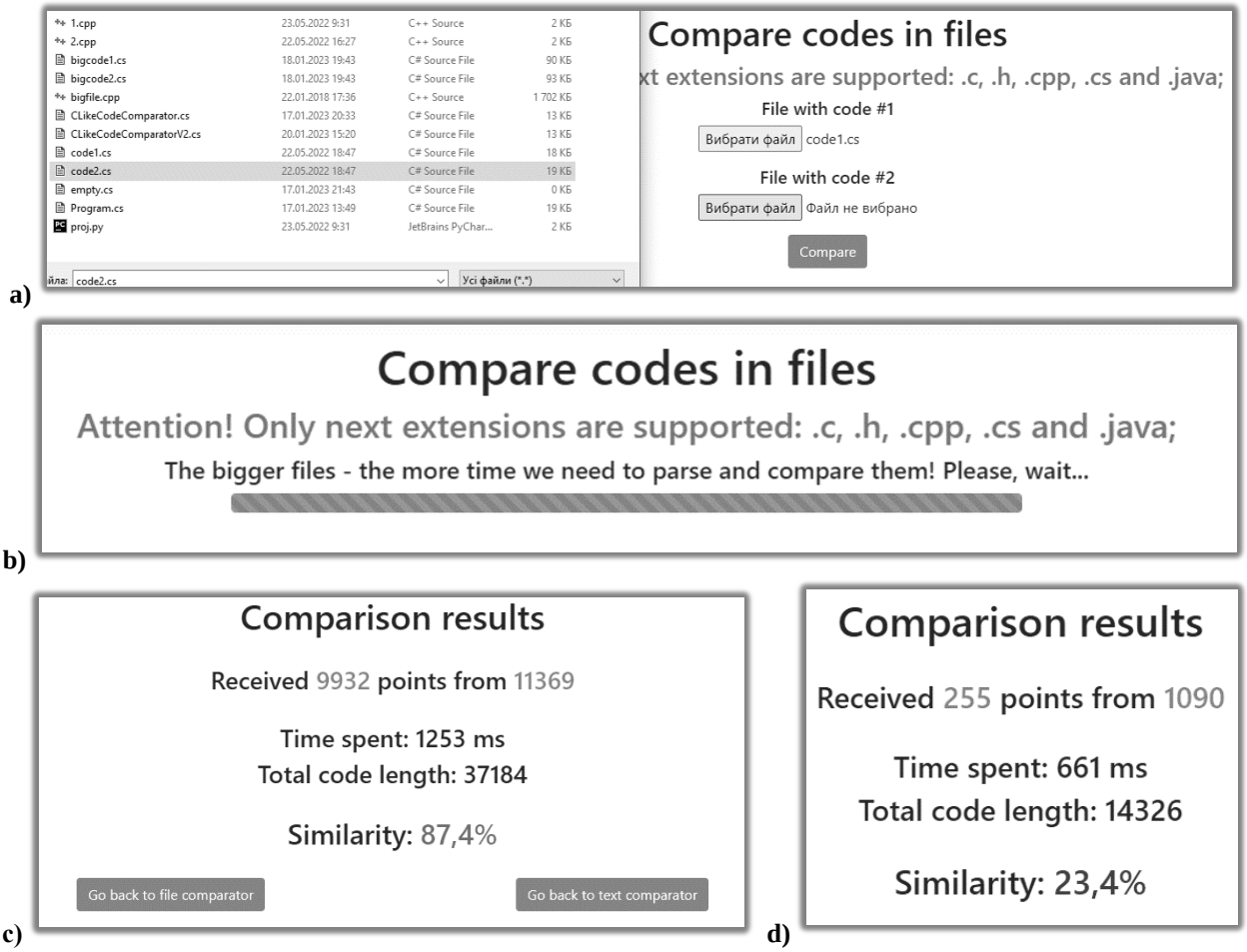

Fig. 6 – application use case: a – file choosing, b – waiting for verification, c – result view, d – alternative result (another files)

Performance analysis can also be performed based on the information provided by the application. It is important to note that the algorithm is quite complex, and for program codes with a size of 500 lines or more, the execution time can be measured in seconds, but the purpose of the algorithm in general is to check program codes - solutions to creative and Olympiad problems. Such solutions are usually not gigantic in size, so even for simultaneous scheduling and checking of a number of solutions, the program runs fast enough (Table 1).

Table 1 – Approximate waiting time for results

Set of tests	Size of code (for 2 files summary, lines)	Size (for 2 files summary, characters)	Time spent (ms)
Set #1	~100	~3000	~12
Set #2	~400	~12500	~215
Set #3	~800	~32000	~1215
Set #4	~2000	~ 92700	~10850

V. CONCLUSIONS

The developed application can be modified in various ways - improved and brought to the form of a full-fledged software product or integrated into an already existing application as its extension, but its basis, the work algorithm, will remain relatively unchanged. The developed models work with code presented as a string data type, therefore, depending on how the data will be received by the application, it can be converted to this type before using the classes (for example, read the code as one line from a file) or directly transferred to execution. Also, the application can be specified for other programming languages by adding a new decomposer, which will be specified for this language or a set of programming languages similar in syntax while the principle of the decomposer will not change - only some symbols and the reaction of the decomposer to them, operations with pointer will be changed. The application, improved in one way or another, can be useful for higher education institutions as a way to partially automate the process of checking the creative works of students majoring in Information Technology, and/or integrated to check solutions to problems at student programming contests.

As an example, after checking the code for a solution to a particular problem, if this is first solution or check result does not exceed a given starting value (e.g., 90%), then the code or reference to it can be saved. Next time, when checking a new solution, it will be decomposed and compared with other more or less unique solutions, saved earlier.

Список використаних джерел

1. Plagiarism Checker. *Dupli Checker* : вебсайт. URL: https://www.duplichecker.com (дата звернення: 12.12.2022).
2. Plagiarism Checker. *Plagiarism Checker.co* : вебсайт. URL: https://www.plagiarismchecker.co (дата звернення: 12.12.2022).
3. CODE DIFF. *W3docs* : вебсайт. URL: https://www.w3docs.com/tools/code-diff/ (дата звернення: 12.12.2022).
4. Compare text. *Diffchecker* : вебсайт. URL: https://www.diffchecker.com/text-compare/ (дата звернення: 12.12.2022).
5. Кірей К. О. Алгоритми та структури даних: методичні рекомендації для виконання лабораторних робіт студентами денної форми навчання спеціальності 121 «Інженерія програмного забезпечення». Миколаїв: Вид-во ЧНУ імені Петра Могили, 2019. 90 с.
6. The vouchers. Eolymp, 2023. URL: https://www.eolymp.com/en/problems/6 (accessed Apr. 02, 2023)
7. Hall B. Beej's Guide to C Programming. Self Publishing, 2022. 749 p.
8. Szuhay J. Learn C Programming: A beginner's guide to learning the most powerful and general-purpose programming language with ease. Birmingham: Packt Publishing, 2022. 741 p.
9. Long S. Learn to Code with C. Raspberry Pi Press, 2020. 92 p.
10. Wisnu A. C++ Data Structures and Algorithms. Packt Publishing, 2018. 322 p.
11. Francesco A., Gabriel B. Software Architecture with C# 9 and .NET 5. Packt Publishing, 2021. 654 p.
12. Albahari J., Albahari B. C# 10 Pocket Reference. O'Reilly, 2022. 270 p.
13. Bancila M. Modern C++ Programming Cookbook. Packt, 2020. 751 p.
14. Horstmann Cay S. Core Java for the Impatient. Pearson Education, Inc., 2023. 884 p.
15. Raza M. Rashid. Getting Skilled with Java (Code). BPB Publications, 2022. 254 p.

References

[1] Dupli Checker. (2023). *Plagiarism Checker* [Online]. Available: https://www.duplichecker.com
[2] Plagiarism Checker.co. (2023). *Plagiarism Checker* [Online]. Available: https://www.plagiarismchecker.co
[3] W3docs. *CODE DIFF* [Online]. Available: https://www.w3docs.com/tools/code-diff
[4] Diffchecker. ()2023. *Compare text* [Online]. Available: https://www.diffchecker.com/text-compare
[5] K. O. Kirei, *Alhoritmi ta strukturi danikh: metodichni rekomendatsiyi dlya vikonannya laboratornikh robit studentami dennoyi formi navchannya spetsialnosti 121 «Inzheneriya prohramnoho zabezpechennya»* Mikolayiv: Chornomorskiy natsional`niy universitet imeni Petra Mohili, 2019.
[6] Eolymp. (2023). *The vouchers* [Online]. Available: https://www.eolymp.com/en/problems/6
[7] B. Hall, *Beej's Guide to C Programming*, Self Publishing, 2022.
[8] J. Szuhay, *Learn C Programming: A beginner's guide to learning the most powerful and general-purpose programming language with ease*, 2nd ed. Birmingham: Packt Publishing, 2022.
[9] S. Long, *Learn to Code with C*, Raspberry Pi Press, 2020.
[10] A. Wisnu, *C++ Data Structures and Algorithms*, Packt Publishing, 2018.
[11] A. Francesco and B. Gabriel, *Software Architecture with C# 9 and .NET 5*, 2nd ed. Packt Publishing, 2021.
[12] J. Albahari and B. Albahari, *C# 10 Pocket Reference*. O'Reilly, 2022.
[13] M. Bancila, *Modern C++ Programming Cookbook*, Packt, 2020.
[14] H. Cay S., *Core Java for the Impatient*, 3rd ed. Pearson Education, Inc., 2023.
[15] Raza M. Rashid, *Getting Skilled with Java (Code)*, BPB Publications, 2022.

Отримана в редакції 01.11.2023. Прийнята до друку 04.12.2023. Received 01 November 2023. Approved 12 December 2023. Available in Internet 03 January 2024