Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.М07-мм

Ван Тяньцзин

Средство автоматической проверки выполнения заданий по программированию с анализом качества кода и поиском плагиата

Отчёт по производственной практике

Научный руководитель:
Доцент кафедры системного программирования, к. ф.-м. н. Д. В. Луцив

Санкт-Петербург
2023

Saint Petersburg State University

Tianjing Wang

Master's Thesis

Toolkit for automatic checking of programming assignments completion with code quality analysis and plagiarism search

Education level: master

Speciality *09.04.04 «Software Engineering»*

Programme *CB.5666.2021 «Software Engineering»*

Scientific supervisor:
C.Sc., docent D. V. Luciv

Reviewer:

Saint Petersburg
2023

Contents

Introduction

With the development of information technology, especially the the Internet, it has become more convenient and faster for people to obtain information resources, and at the same time plagiarism has become easier. In programming courses, homework is stored in the form of electronic documents, so students can directly copy homework from the Internet or other students, and hand it to the teacher directly with simple modification or without any modification. Plagiarism makes students develop lazy habits, and some students will get nothing. To curb plagiarism, we must severely punish those students who plagiarize. First of all, we must find students who plagiarize. But checking for plagiarism will be quite a heavy workload if the teacher manually compares in a large number of program assignments. The program code plagiarism detection system can compare each pair of program codes in a large number of program assignments, obtain the similarity of each pair of programs, and display the results to the user. These results can help teachers find out the assignment suspected of plagiarism and provide certain references for further judging plagiarism. The use of this system will greatly improve the efficiency of teachers' plagiarism detection and save a lot of working time.

In addition, in the field of software business, if one party thinks that the other party has plagiarized the internal core technology of its product, an efficient plagiarism detection system is used to test the two software products, and the result will be of great significance to the final identification is of great help. At present, with the increasing emphasis on intellectual property rights and the strengthening of the protection of software products, the research has also attracted more and more attention.

The source code of a program can be regarded as text with some special specifications, so the program code can be treated as a continuous *token* string. In program code plagiarism detection, the program can be converted into *token* strings that retain important features of the program according to a certain feature extraction method, and then similarity information can be obtained by matching these *token* strings. In this way, the time

for comparing program source codes can be reduced, and the detection efficiency can be improved.

In the above application fields, string matching algorithms play an important role. String matching is mainly used to solve the problem of pattern search. However, traditional pattern search cannot solve the search problem of finding all similar parts in two texts. In order to apply the traditional string matching algorithm to the program code *token* strings similarity detection, it is necessary to make appropriate changes and improvements. Therefore, researching and realizing the string matching algorithm in program plagiarism detection technology has important practical significance for developing a system that assists teachers in plagiarism detection, and also has certain reference value for other application fields related to string similarity judgment.

Meanwhile, the string matching problem is a fundamental problem in computer science and one of the most widely studied problems in complexity theory. It has a wide range of applications in Text Editing Processing, Image Processing, Document Retrieval, Natural Language Recognition, Biology and other fields. Moreover, string matching is the most time-consuming core problem in these applications, and a good string matching algorithm can significantly improve the efficiency of the application.

Research on program plagiarism detection technology started relatively early, beginning in the 1970s. At present, there are many effective plagiarism detection systems, such as: JPlag [7], MOSS [1, 13] and YAP [16] and so on. They have been successfully applied to plagiarism detection of student programs and plagiarism detection of documents.

The final realization of the plagiarism detection system achieves the purpose of assisting manual judgment by reducing the time spent comparing program texts.

At present, the existing plagiarism detection system judges the plagiarism of programs or documents based on comparing the similarity of programs or documents. The more similar they are, the greater the possibility of plagiarism between them. The quantified result is the similarity. Most plagiarism detection systems will give this value. Generally speaking, the

greater the similarity, the greater the possibility of plagiarism.

Scholars from various countries have done some research on string matching algorithms used in program plagiarism detection technology, among which RKR-GST algorithm has been successfully used in multiple detection systems (Jplag, YAP, etc.). At the same time, the algorithm can also be used to detect the similarity of biological sequences. Compared with other algorithms used to detect string similarity, this algorithm has the advantages of high detection efficiency and fast speed. Experiments show that the average time complexity of this algorithm is close to linear.

1 Problem statement

The goal is to implement the toolkit to assist teachers or trainers engaged in software teaching to automatically check students' programming assignments through source code quality analysis and plagiarism search.

- Conduct the survey of existing techniques, tools and algorithms used for plagiarism detection.
- Propose the solution for quick interactive code plagiarism detection and code quality check.
- Propose the architecture of above solution.
- Design and implement plagiarism detection algorithm.
- Implement the solution.
- Test the solution using real code examples.

2 Background of study

2.1 Techniques and tools

Program code plagiarism detection is an important part of text plagiarism detection field. The so-called text plagiarism detection is to judge whether a given document is plagiarized or copied from the content of another document or documents. Plagiarism not only means copying the original text, but also includes shifting and transforming the original work, synonyms Replace and change the way of restatement and so on. Text plagiarism detection includes two categories: program code plagiarism detection and natural language text plagiarism detection. Natural language text plagiarism detection is mainly for text data, especially academic papers, while program code plagiarism detection is mainly for program code. The main research purposes of program code plagiarism detection are computer-assisted teaching and software intellectual property protection. Program code plagiarism detection mainly calculates the similarity between two program codes to obtain a similarity value (generally a value between 0.0 and 1.0), and then sets a threshold to determine whether there is plagiarism in the target software or program code assignment. From the 1980s to the present, researchers have been conducting research on program code plagiarism detection technology. At present, commonly used program code plagiarism detection techniques are mainly divided into two categories: **Attribute Counting** and **Structure Metrics**.

Attribute counting. Attribute counting method mainly performs statistical processing on various attributes in the program code, maps these attributes to the vector space, and then calculates the similarity between the two. The software scientific measurement method [9] is the earliest attribute counting method. Firstly, the measurement standard of software similarity is given. According to the standard, multiple software measurement characteristics are defined, and then the software measurement characteristics contained in the program code are counted to generate the corresponding feature vector. Finally, the cosine metric formula is used to

calculate the similarity between two vectors as the similarity between two software. Most of the code plagiarism detection technologies based on attribute counting methods are researched on the basis of software scientific measurement methods. In 1996, on the basis of the software scientific measurement method, Sallies et al. [12] considered capacity, control flow, data dependence, nesting depth, and control structure six parts of attributes when statistical software measurement characteristics, and formed a six-tuple vector array. The similarity calculation is then performed on the six-tuple vector. Experiments show that the detection effect of this method is better than that of the software science measurement method, but the detection accuracy is still poor, and there are many cases of misjudgment. Some researchers also propose to improve the detection accuracy by increasing the dimension of the feature vector on the basis of the software scientific measurement method, but the experimental results prove that the effect is not obvious. Verco et al. also pointed out that "simply increasing the dimension of the vector cannot reduce the error rate of detection and improve the detection accuracy" [14]. Therefore, adding more program code structure information and semantic information in the detection process can fundamentally improve the shortcomings of the attribute counting method and improve the detection accuracy.

Structure Metrics. Compared with the attribute counting method, the structure measurement method adds more internal structure information and implicit semantic information of the program to the detection process. Conduct in-depth analysis to generate a data sequence that can represent the meaning of the program code, and then perform similarity calculations on the data sequence. The detection accuracy of the structure measure method has been improved compared with the attribute count method. Detection methods based on structural metrics usually include the following two steps:

(1) Program code formatting, that is, converting the program code into a standard format. For example: Convert identifiers in program code to specific symbols, filter out blank lines, blank characters and comment statements in code, unify uppercase and lowercase letters, etc. There are

many formatting methods, and the methods currently used by researchers include: **String-based methods, Token-based methods, Tree-based methods, Semantic-based methods**, etc.

String-based method: First, divide the program code into strings by line, and each program fragment contains adjacent strings; then, judge whether the strings between two program fragments are the same, if they are the same, then The two program fragments are similar; otherwise, they are not similar; finally, it is determined whether there is plagiarism between the two program codes according to the similarity of the program fragments contained in the program codes. A more representative String-based detection method is the parameterized matching algorithm proposed by Baker in 1995 [2, 3]. This algorithm uniformly formats the identifiers and literals in the program code and then performs similarity comparison. However, after the unified formatting, the detection There will be a large deviation in the result, and the detection accuracy is not high.

Token-based method: Perform lexical analysis on the program code to generate a Token sequence, and then detect the same Token sequence fragment in the Token sequences generated by the two program codes. Compared with the String-based method, the Token-based method is more robust to the detection of formatting codes and code gaps, and its detection efficiency is very high, but the detection accuracy is still poor.

Tree-based method: Perform lexical and grammatical analysis on the program code, and obtain the corresponding abstract syntax tree. If the two subtrees contained in the two abstract syntax trees are the same or similar, the two subtrees are judged to be similar Subtrees, and then judge the similarity of programs according to the similarity of similar subtrees [2, 4, 6] contained in the abstract syntax tree. Compared with the String-based method and the Token-based method, the detection accuracy of the Tree-based method has been greatly improved, but due to the large redundancy of the abstract syntax tree and the high optimization cost, the detection efficiency of the Tree-based method is relatively low. Difference.

Semantic-based method: Komondoor et al. first converted the program code into a program dependency graph (PDG) [11], and then used

program slicing technology (Program Slicing) [15] to determine whether the subgraphs of the two program dependency graphs are the same or isomorphic, thereby determining whether the two Whether the program code is suspected of plagiarism. The method based on Semantic has high detection accuracy, but the time and space complexity is very high, it is difficult to detect the program code with a large amount of data, and it cannot be practically applied.

(2) Similarity calculation, that is, to calculate the similarity of the data sequence obtained after the program code is formatted, and calculate the similarity value between two data sequences. Commonly used methods include vector space model method and string matching algorithm, including: dot matrix method, Levenshtein distance formula, cosine metric method, sequence matching algorithm, longest common subsequence algorithm, GST algorithm and RKR-GST Algorithms and more.

Whether it is attribute counting method or structural measurement method, any single detection method has its own shortcomings and adaptability problems. Since the attribute counting method does not take into account the internal structure information of the program, it is only necessary to slightly modify the program code structure to detect plagiarism in the program code, and the detection accuracy is low. The structure measurement method adds more program structure information in the process of program code detection, and the detection accuracy has been improved a lot compared with the attribute count method, but it still cannot detect some complex problems without in-depth analysis of the data flow and control flow of the program. means of plagiarism. In order to better balance the detection efficiency and detection accuracy, most of the program code plagiarism detection systems developed in recent years combine the attribute counting method and the structural measurement method, such as: JPlag system, MOSS system, Sim system [8], YAP3 system, CCFinder system [10], CloneDR system [2, 6] and CP-Miner system [5], etc. Among them, MOSS, YAP3 and JPlag systems are the most widely used. The core algorithm of MOSS system is Winnowing algorithm, and both YAP3 and JPlag adopt RKR-GST algorithm. Most systems will eventually return a

value between 0.0 and 1.0 as the similarity between two program code pairs, and will set a threshold (the threshold can be automatically set by the system or selected by the user), using It is used to divide the program code pairs suspected of plagiarism. When the similarity value between two program codes exceeds a given threshold, plagiarism is suspected between the two. This creates a problem: the threshold is difficult to determine. Because the threshold is too large, some program codes suspected of plagiarism may be missed; the threshold is too small, it is easy to cause misjudgment, and the program code that does not contain plagiarism is judged as suspected of plagiarism. Most of the thresholds are obtained through a large number of experiments. When the type and scale of the detection data are quite different, it is impossible to use a fixed threshold to divide the detection results. Most of the existing program code plagiarism systems can only detect part of the plagiarism means. When the program code contains some redundant variables, statements, or some expressions or statements are split or reordered, the detection accuracy of the detection system will decrease. Obvious reduction. In addition, when the code size of the program code is large, although the similarity value between the two program codes is low, there is still a suspicion of plagiarism between the two program codes. For example: the number of lines of the two program codes is about 1000, and there are 300 lines of codes that are similar. Although the calculated similarity is only about 0.3, there is still suspicion of plagiarism between the two program codes; or the two program codes There are exactly the same 150 lines of continuous code in , and the similarity may be less than 0.2, but these two program codes can still be identified as suspected plagiarism. The type and size of the program code have an important influence on the determination of the threshold, and the selection of the similarity threshold is very important in the plagiarism detection of the program code.

From the early 1980s to the present, researchers at home and abroad have been conducting research on program code plagiarism detection technology, and have developed a number of representative program code plagiarism detection systems. The following will introduce and analyze several typical program codes Plagiarism detection system.

2.1.1 Plague system

In 1988, G. Whale developed the Plague system. This system mainly uses the detection method based on the structure measurement method, and uses the internal structure information contained in the program code to detect the plagiarism of the program code. The detection mainly includes three stages:

- Convert each program code file into a structural metric table and a signature sequence describing its structural features. Among them, the structural metrics table mainly includes the structural features in the program code, such as: selection, loop, conditional statement and function, etc., and is described by regular expressions. The feature tag sequence contains all the important features in the program code mark.

- Use a specific function to compare the similarity between the structural feature tables generated by the two program codes, filter out the program pairs with low similarity, and leave the program pairs with high similarity to the next stage for processing.

- Use the improved longest common subsequence algorithm to calculate the similarity of the signature sequences of the remaining program pairs to obtain the detection results.

The defects of the Plague system mainly include the following points:

- The Plague system can only detect PASCAL, Shell, Prolog and Llama, four programming languages, it is not applicable to other programming languages, and it is difficult to extend to other programming languages.

- The detection accuracy of the Plague system is not high, and the detection results are not clear at a glance, because its result is a list sorted by two indexes, which requires further explanation by professionals.

- The detection efficiency of the Plague system is not high.
- Plague system has poor portability.

2.1.2 Sim system

Dick Grune proposed the Sim (Software Similarity Tester) [8] system in 1999, which is mainly used to detect the similarity between program codes written in C, Java, Pascal and other programming languages. The system directly uses a string alignment algorithm for detecting DNA sequence similarity, and finally returns a value between 0.0 and 1.0 as the similarity result between the two program codes. The running time complexity of the Sim system is $O(n^2)$, n is the maximum length of the syntax analysis tree generated when the system is initialized. The detection steps of the Sim system are as follows:

- A token string that can represent the structure and meaning of the program code is generated by a lexical analyzer. Each token can represent a keyword, a number, a string constant, a comment, an identifier, or a function, etc. Sim is highly scalable and can be easily extended to other languages.
- After converting all the program code pairs into corresponding tag strings, divide the tag string generated by one of the program codes into several parts, each part represents a module of the program code, and then in the tag string of the other program code Match each module. This allows for better detection of plagiarism, both types of typography and code block reordering. Corresponding scores will be given after each module matching, and different matching levels will give different scores, such as: 2 points for two modules matching, 0 points for two modules not matching, etc.
- Use the score obtained by module matching to calculate the similarity of the program, and normalize it to between 0.0 and 1.0. The calculation formula is as in formula:

$$Similarity = \frac{2 \times score(p1, p2)}{score(p1, p2) + score(p2, p2)} \quad (1)$$

Among them, Similarity indicates the similarity between program codes, and score() indicates the matching score of two modules.

The Sim system works well when used to detect plagiarism in small computer coursework, but the Sim system can only detect part of the plagiarism methods, especially when the program code is modified a lot and the structure changes greatly, the detection accuracy of the Sim system is obvious decline.

The Sim system is used to detect the number of codes to be programmed and the average code size is small, and the running speed is very fast. For example, it takes 3.5 minutes to detect the similarity between 56 program codes with an average length of 3415 bytes. However, when the average code size of the program code set to be tested is large and the number is large, the detection efficiency of the Sim system will be greatly affected, and the running speed will be significantly reduced.

2.1.3 YAP series

The YAP (Yet Another Plague) series system is developed by Wise on the basis of Plague. There are three versions in total. The first version is YAP1 developed in 1992, the second version YAP2 was launched two years later, and finally the first version was launched in 1996. Three editions of YAP3. The YAP1 and YAP2 systems can only be used to detect program code plagiarism detection, while the YAP3 system can detect both program code plagiarism and natural language text copying. When the YAP series system detects program code plagiarism, it also measures the similarity of two program codes, and finally gives the matching ratio (from 0 to 100) between the two programs as the similarity.

The program code plagiarism detection process of the YAP series can be roughly divided into two stages:

- Perform lexical analysis on the program code, remove the program code that does not affect the program structure and semantics, and

convert the program code into a specific token (Token) string. The main operations include: delete comments in the program code; delete all illegal identifiers in the program code; unify uppercase and lowercase letters in the program code; replace synonyms and synonymous library functions with the same words and functions.

- Carry out similarity calculation. The three versions of the YAP series use different similarity calculation algorithms: YAP1 combines the two methods of Unix general program and command program script; YAP2 uses the Heckel algorithm; YAP3 uses RKR-GST (Running Karp Rabin - Greedy String Tiling) algorithm, the RKR-GST algorithm searches for the largest matching string in the token strings generated by the two program codes, and uses the length of the largest matching string as the basis for similarity measurement.

The YAP series only compares token strings composed of keywords in programming language dictionaries, and does not perform comprehensive syntax analysis on program codes. The YAP3 system has the best detection effect in the YAP series. The YAP3 system [**?**] can detect most of the 12 types of program code plagiarism methods, including complete copying, modifying comments, adding or modifying blank lines, modifying identifiers, changing data types, changing the order of operands in expressions, code Block reordering, reordering of statements in code blocks, adding redundant statements or variables, etc., but the detection of some of these plagiarism methods will be biased. Wise has proved through repeated experiments that the YAP3 system works well when used to detect plagiarism in computer program assignments.

2.1.4 MOSS system

Dr. Alex Aiken of Stanford University developed the MOSS (Measure of Software Similarity) system [**?**, **?**] in 1994, which is mainly used to detect possible plagiarism in program code, and supports C, C++, Java, PASCAL, ML, Ada, Lisp, Scheme, etc. language. In 2004, Dr. Alex Aiken

improved the system and added support for some programming languages. The programming languages supported by the latest version of the MOSS system include: C, C++, Java, Visual Basic, Javascript, FORTRAN, ML, Haskell , Lisp, Scheme, Pascal, Python, Perl, Matlab, Modula2, Ada, TCL, VHDL. Verilog, Spice, etc. The MOSS system provides users with web services. Users can submit program code text sets to be tested through script programs. After plagiarism detection, the system will return the detection results to users in the form of Web pages.

In order to prevent plagiarists from evading plagiarism detection by being familiar with the detection principle of the MOSS system, Dr. Alex Aiken did not give a detailed algorithm for similarity calculation in the MOSS system, but only gave a rough algorithm idea, and proved that the MOSS system is better than the statistics program. A system with specific word frequencies is more effective [**?**, **?**]. The detection algorithm of the MOSS system combines the attribute counting method and the structure measurement method, and includes data preprocessing, statistical identifier occurrence times and string matching algorithms in the detection process. Among them, the algorithm idea of the most critical string matching algorithm is roughly divided into the following four steps:

(1) Divide each program code to be detected into contiguous substrings of length k. The size of the parameter k can be defined by the user, or the system default setting can be used.

(2) Use a specific hash function to hash each substring of length k to obtain a hash set composed of all substrings.

(3) Select a subset that can represent the meaning of the program code from the generated hash set as the program fingerprint of the program code.

(4) Calculate the similarity of the program fingerprints generated by two different program codes, and obtain the similarity value between the two program codes as the detection result.

In the MOSS system, the more substrings that match each other in the substring sets generated by two program codes, the greater the similarity between the two program codes. The MOSS system is used to detect plagiarism in computer program course assignments with good results. However,

when there is a big difference between the amount of code and the size of the code in the program code set to be detected, for example: one of the program codes to be detected has a larger amount of code, while the other has a smaller amount of code.

The detection accuracy of the MOSS system will be greatly affected. All the processing of the MOSS system is carried out in the main memory, the detection efficiency is not high, and it is difficult to handle large-scale data.

2.1.5 Jplag system

The JPlag system [**?**] was developed by L.prechelt and others at the University of Karlsruhe in Germany in 1996 with the Java language. It provides program code plagiarism detection services on the Internet. This system is mainly used to detect possible plagiarism in program codes written in languages such as Java, C, C++ and Scheme. The similarity calculation method used is the same as YAP3, which is also the RKR-GST algorithm, but the detection algorithm of the JPlag system is optimized. The time complexity of the RKR-GST algorithm in YAP3 improves the detection efficiency.

The program code plagiarism detection algorithm of the JPlag system is divided into the following two steps:

(1) Generate a corresponding token string (token string) for each program code.

(2) Divide the token string generated by each program code into smaller substrings called "tiles", and finally perform similarity calculations on the generated substring sets, and calculate the proportion of successfully matched strings The ratio of all substrings is converted into a value of similarity between the two programs.

The similarity calculation formula is shown in formula:

$$Sim(A, B) = \frac{2 \times Coverage(tiles)}{|A| + |B|} \tag{2}$$

$$Coverage(tiles) = \sum_{match(a,b,length) \in tiles} length \tag{3}$$

Among them, $|A|$, $|B|$ represent the length of the mark string corresponding to the program code in the program code file A and B: a, b represent the start position of the mark string; the $match(a, b, length)$ function represents the start The same substring at positions a, b, and length length.

When the program code plagiarism detection is completed, the system will return to the user a page including program details, the user can view the directory name of the program, the language used by the source program, the file extension, the number of compared files, etc., and when the user views When there are program pairs with high similarity, the system will highlight the similar program codes in the program pair. It has been verified by repeated experiments that JPlag is as powerful as MOSS and YAP3 in many aspects.

2.2 String matching algorithm

String (that is, string) is an important data structure. The object of computer non-numerical processing is often string data, such as in assembly and high-level language compiler, source program and target program are both string data. In text editing problems it is often necessary to find all occurrences of patterns in a text. A typical example: find the word (pattern string) that the user needs in the text. String matching (String matching) is to find out where a certain pattern (pattern) appears in a certain text (text) (if it can appear there). This problem can be applied in application fields such as keyword (word) search and similarity comparison. An efficient string matching algorithm can greatly improve the efficiency of solving this problem. In recent years, with the expansion of the application field of string matching algorithm, the research on string matching algorithm has become more and more in-depth.

2.2.1 Description of String matching

The string matching problem is described as follows: Assume that the text string T is described as a character array $T[1...n]$ of length n, and the sample string P is described as a character array $P[1...m]$ of length m, and $m \leq n$. Let each element of the strings T and P be in the set $\sum$. For example, if both T and P are English character strings, then $\sum = a, b, ..., z$. For a position $s(0 \leq s \leq n-m)$ in the T string, if $T[s+1...s+m] = p[1...m]$, then s is called a $validshift$. The string matching problem is to find out that all T strings contain the start bit of P string, that is, the effective shift s process. For example: $T = abcabaabcabac$, $P = abaa$ then applying the matching algorithm will get $s = 3$ 1.

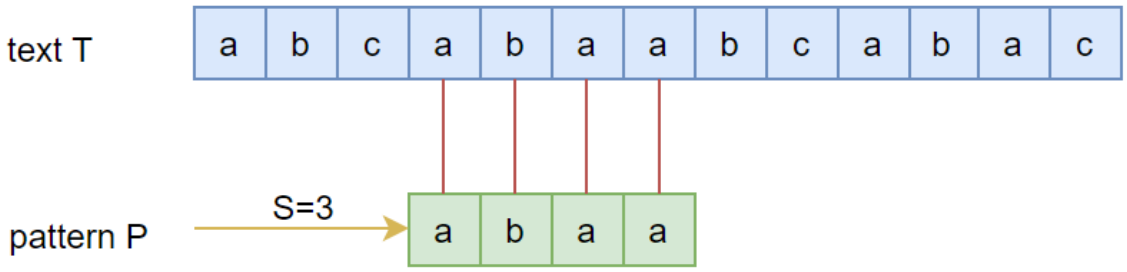

Figure 1: String pattern matching

2.2.2 Application of String Matching

String matching technology has a wide range of applications, and can be applied in information retrieval, network security (such as intrusion detection, virus detection), gene detection and other fields.

Network Intrusion Detection: To prevent unauthorized operation of computer systems, or access to data and other resources stored within the system. By listening to the detailed information about user activities recorded in the system log, we can collect possible "evidence" of illegal intrusion into the system by the user, so as to judge whether the user's behavior is legal. The main job of an intrusion detection system is to match the security rules of the system in the monitored data, and take corresponding

steps according to the found intrusion situation. The string matching technique is explicitly applied here and becomes a decisive factor affecting the performance of an intrusion detection system.

Biological Science: In the field of biological sciences, look for the position of a gene fragment or a group of gene fragments in a gene sequence to compare the similarity and genetic relationship of genes, or to understand whether a gene has a disease, etc. Because genes can be represented by symbol sequences based on a certain character set, this search operation can be realized by string matching algorithms.

2.2.3 Status of String Matching

According to different applications of string matching, string matching algorithms can be divided into: perfect string matching (Perfect String Matching) and approximate string matching (Approximate String Matching). The function of the exact matching algorithm is to find the occurrence position of a substring that is completely equal to one or a set of specific pattern strings in the data sequence; the function of the approximate matching algorithm is to find all the substrings in the data sequence according to a certain similarity measure. All substrings whose similarity to a specific pattern string or a set of pattern strings is within a certain range. Exact matching algorithms are mainly used in application fields such as text retrieval and network security; approximate matching algorithms are mainly used in application fields such as biology and signal processing.

Exact String Matching: The exact string matching algorithm (Perfect String Matching) is the process of finding the substring position equal to the pattern string in the main string. If it is found, the match is said to be successful, and the function returns the storage location (or serial number) of the first occurrence of the pattern string in the main string, otherwise, the match fails and -1 is returned. Commonly used string matching algorithms include: **Brute-Force matching algorithm**, **Rabin-Karp** and **Knuth-Morris-Pratt (KMP)** and other algorithms. The time required for each algorithm can be divided into two parts: the time spent in preprocessing and the time spent in the matching process.

- **Brute-Force matching algorithm** (also known as B-F algorithm or naive algorithm) is suitable for small-scale string matching. The basic idea of the algorithm is: compare the first character of the main string $s =$ "$s_0 s_1 ... s_{n-1}$"with the first character of the pattern string $t =$ "$t_0 t_1 ... t_{n-1}$", if they are equal then Continue to compare subsequent characters; otherwise, start from the second character of the main string s to compare with the first character of the pattern string t; and so on. If each character in the pattern string t is equal to a continuous character sequence in the main string s, the pattern matching is successful, and the function returns the subscript of the first character of the pattern string t in the main string s; if the main string is compared If there is no substring equal to the pattern string t in all character sequences of the string s, the pattern string matching fails, and the function returns -1. This algorithm does not need preprocessing, and the time used in the matching process is $O((n - m + 1)m)$.

- **Rabin-Karp** algorithm is: define a hash function (or fingerprint function), find the hash value (fingerprint value) corresponding to the pattern, and only those substrings with the same hash value as the pattern in the text are possible Matches the pattern string, so it is not necessary to examine all substrings of length m in the text at the same time. Only if the hash values of the two are equal are the patterns and the body substrings actually matched on a character-by-character basis. The algorithm needs to be preprocessed first, that is, to generate a hash value. The time complexity of this process is $O(m)$, and the time complexity for matching is $O((n-m + 1)m)$. Although the time complexity is the same as that of the B-F matching algorithm, its average time complexity is close to linear.

- **KMP** algorithm was proposed by Knuth et al. It has made great improvements to the simple algorithm, mainly because the search pointer does not need to backtrack every time a certain match fails, but uses the obtained "partial match" results to move the pattern to the right. Slide" several positions (put in a next array) and continue

the comparison. The algorithm also requires preprocessing, and the time complexity is $O(m)$. The time complexity required for matching is $O(n)$.

Approximate String Matching: Approximate String Matching (Approximate String Matching), also known as a fault-tolerant pattern matching algorithm, is an important variant and extension of the string matching problem. The matching is to find the substring matching the pattern string in the text string according to some approximate standard. The Multiple Approximate String Macthing (Multiple Approximate String Macthing) is to find the substrings in the text string that match the pattern string set according to some approximate standard. Approximate string matching can be described as: Given a target text T of length n, a pattern P of length m, and a maximum allowable error $k(k < m)$, what we need to do is to search in the target text T for matching conditions , so that the substring undergoes at most k times of editing operations such as replacement, deletion, and insertion, which are the same as pattern P. Classic approximate string matching algorithms include: dynamic programming algorithm, automaton algorithm, filtering algorithm and bit parallel algorithm, etc.

2.2.4 Evaluation Criteria for String Matching

In general, a good string matching algorithm should have the following characteristics:

Fast speed: This is the most important criterion for evaluating a character matching algorithm. It is generally required that character matching can be performed at linear speed. There are several time complexity evaluation metrics:

- Complexity of preprocessing time: Some algorithms need to preprocess pattern features before string matching.
- Time complexity of the matching stage: the time complexity of performing the search operation during the string matching process,

which is usually related to the length of the text and the length of the pattern.

- Worst-case time complexity: When performing character pattern matching on a text, trying to reduce the worst-case time complexity of each algorithm is one of the current research hotspots.
- Time complexity in the best case: the best possibility when character pattern matching is performed on a text.

Less memory usage: Executing preprocessing and pattern matching requires not only CPU resources but also memory resources. Although the current memory capacity is much larger than before, in order to increase the speed, people often use special hardware. Usually, the memory access speed in special hardware is fast but the capacity is relatively small. At this time, the algorithm that occupies less resources will be more advantageous.

3 Proposed solution description

This section describes the high-level details of the proposed solution. It highlights design principles and package dependencies.

3.1 Design Principles

Plagiarism detection tools are designed to relieve teachers or trainers of the burden of checking students' programming assignments. It is designed the way to maximize the accuracy of detection, achieve linear complexity and occupy the least space.

3.2 Dependency of code quality check

Regarding the check of code quality, we use Flake8. Flake8 is a tool officially released by Python to assist in detecting whether the Python code is standardized. Compared with *Pylint*, which is currently relatively popular, Flake8 has flexible inspection rules and supports the integration of additional plug-ins. Strong. Flake8 is a package for the following three tools:

- **PyFlakes**: A tool for statically checking Python code logic errors.
- **Pep8**: A tool for statically checking the PEP8 coding style.
- **NedBatchelder's McCabe**: A tool for statically analyzing the complexity of Python code.

3.3 Dependency of code plagiarism detection

The indexing and retrieval technology is realized through the Whoosh of the open source community. The installation of Whoosh is very simple. Whoosh is a full-text indexing and retrieval programming library based on the Python language. The development of Whoosh has absorbed the advantages of many other open source index libraries. Its basic architecture

refers to Lucene based on the Java language. All the implementations in this study are based on Python technology. Although there are many good tool libraries based on Java language, Java language is more "bloated" than Python language, with a huge amount of code, which is not suitable for rapid development. The Python language has concise syntax and highly readable code, which is very important for the rapid development and later maintenance of small and medium-sized projects.

Whoosh has many advantages:

- With a good structure, each module such as scoring module and word segmentation module can be replaced as needed.
- It is completely implemented based on the Python language, without binary packages, which saves the tedious process of compiling binary packages, and the program will not crash for no reason.
- Whoosh is currently the fastest full-text indexing and retrieval library implemented in pure Python.

3.3.1 Use of Whoosh

The first step in using Whoosh is to create an index object. First, define the index schema (schema) and list it in the index as a field:

```
from whoosh.fields import *
schema = Schema(title=TEXT, url=ID, content=TEXT)
```

title, url, and content are the so-called fields, and each field corresponds to a part of the information of the target file to be searched by the index. The above code is the mode of establishing the index, and the index content includes title, url, and content. A field is indexed, which means it can be searched and stored. After slightly modifying the above code, it looks like this:

```
from whoosh.fields import *
```

schema = Schema(title=TEXT(stored=True), url=ID(stored=True), content=TEXT)

Here, in the title and url fields, setting stored to True means that the search results of this field will be returned. The index pattern is established above, and there is no need to repeat the index pattern, because once the index pattern is established, it will be saved with the index. In fact, in the application process, according to different situations, you can also create a class for indexing mode. As follows:

```
from whoosh.fields import SchemaClass, TEXT, KEYWORD, ID, STORED
class MySchema(SchemaClass):
    url = ID(stored=True)
    title = TEXT(stored=True)
    content = TEXT
```

In the above code, title is the name of the field, and the following TEXT is the type of the field. These two describe the index content and lookup object type, respectively. Whoosh provides the following field types for indexing mode:

(1) whoosh.fields.ID: It can only be a unit value, that is, it cannot be divided into several words, and is usually used for things such as file path, URL, date, and classification.

(2) whoosh.fields.STORED: This field is saved with the file, but it cannot be indexed or queried. Often used to display file information.

(3) whoosh.fields.KEYWORD: Keywords separated by spaces or commas can be indexed and searched. To save space, word searches are not supported.

(4) whoosh.fields.TEXT: the text content of the file. Index and store text, and support vocabulary search.

(5) whoosh.fields.NUMERIC: Digital type, save integer or floating point number.

(6) whoosh.fields.BOOLEAN: Boolean class value.

(7) whoosh.fields.DATETIME: Time object type.

After the index mode is established, the index storage directory must also be established. As follows:

4 Architecture

This section introduces the overall architecture and modules of the plagiarism detection tool.

4.1 Input module

Here we detect the source file language as Pythonwen file, so our input module is mainly ".py" file or ".ipynb" file.

4.2 Preprocessing module

After the files to be checked for plagiarism are selected, the first step in the processing flow is the preprocessing module. The source code of the software has a variety of writing styles, and the preprocessing requirements of different programming languages are also different, so the main work that this module needs to do is to perform corresponding preprocessing analysis according to the input source code file format. Among them, there are comments, extra blanks, tabs, and extra line breaks in the Python source file to be detected, all of which need to be deleted, because these information do not affect the detection results. After these steps, only meaningful characters are left in the source code, and then further processing is carried out. After multi-layer preprocessing analysis, only characters meaningful to lexical analysis are left, and then passed to the lexical analysis module.

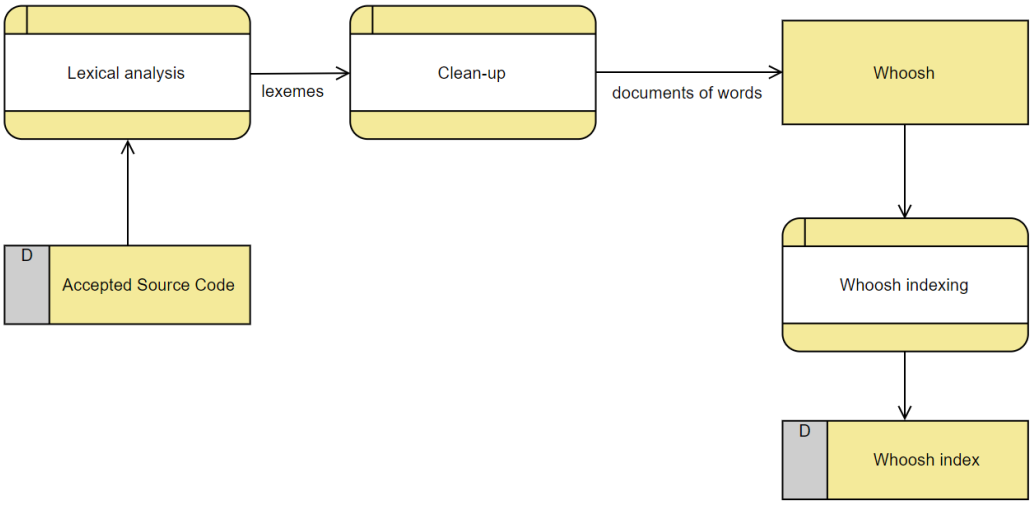

Figure 2: Workflow of indexing

4.3 Lexical analysis

The task of the lexical analysis module is to analyze the input character stream, decompose the character stream into Token streams according to the written matching rules, and then pass them to the Whoosh module to add the index. Lex (Lexical analysis) is mainly used in lexical analysis, and the definition set and rule set of Lex are mainly written by user-defined names and regular expressions.

4.4 Database module

In practical applications, a certain specific software or a certain part of specific source code is often required as a reference file library, and then the source code of different software is compared with the library to further judge the situation of these software

plagiarizing code from the library. And for this tool, we use the Whoosh index repository as our database. The repository contains the benchmark files we need to compare.

4.5 Output module

The output module is output in the form of a ”report”, and the report will display a percentage, which then tells us the ”similarity probability” or ”plagiarism probability” of the two assignments.

4.6 Comparison module

The job of the comparison module is to compare the character strings obtained through the above process between the source code file to be detected and the source code file to be detected. If they are the same, it will be judged as plagiarism; otherwise, there is no plagiarism.

4.7 Components

The tool mainly consists of components such as Plagiarism report generator, Plagiarism detector, Text Cleaner, Search and Python lexeme, and users can access the tool through GUI (Graphical User Interface) and CLI (Command-Line Interfce).

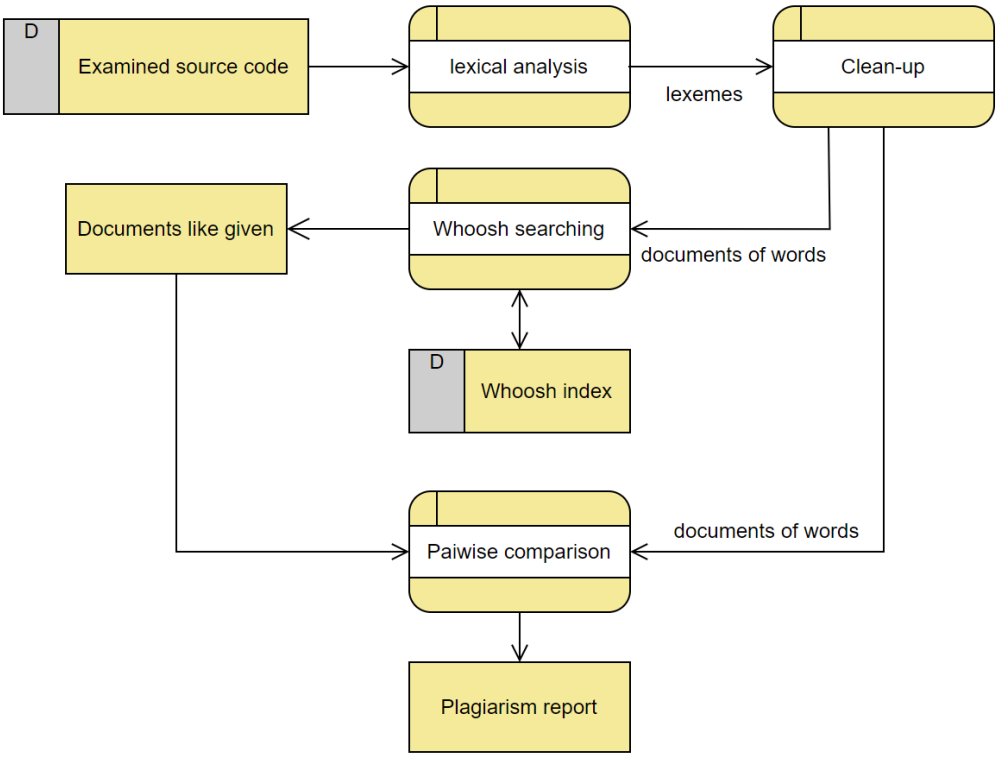

Figure 3: Workflow in search

5 Algorithm implementation

The RKR-GST algorithm was proposed by Wise in 1993, which introduced the exact string matching algorithm — Karp-Rabin algorithm and improved the GST algorithm. Initially, the RKR-GST algorithm was used in the plagiarism detection of computer programs and other texts, such as the plagiarism detection system YAP3 and the Jplag system. Later, it was also used to compare biological sequence differences to find out biological sequences such as mutated amino acids or nucleotides, such as the Neweyes system. This algorithm improves the time complexity of the GST algorithm, and its average time complexity is close to linear through experi-

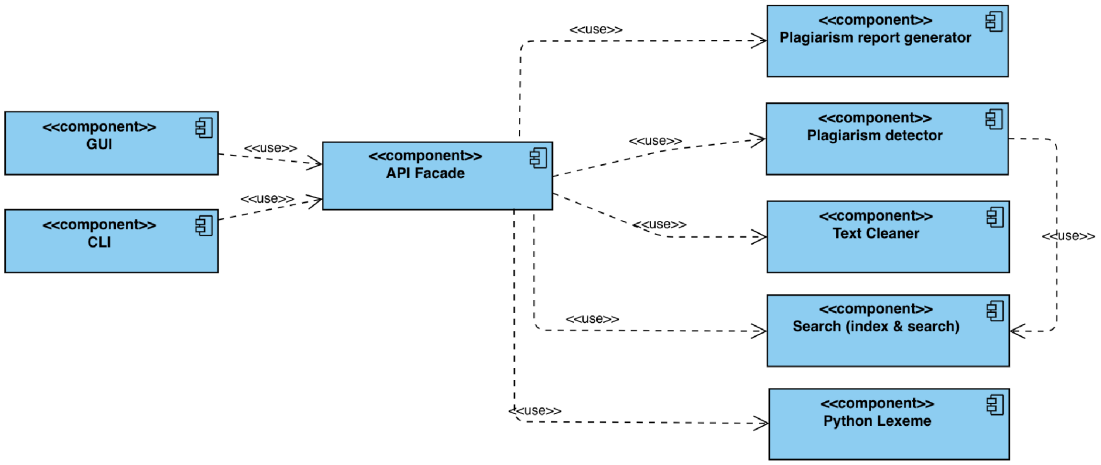

Figure 4: Main components

ments.

5.1 KR string matching algorithm

In 1987, Professor KARP, the winner of the Turing Award, and Professor RABIN, a famous scholar, jointly published an intuitive and fast Karp-Rabin string matching random algorithm in IBM Research and Development Magazine.

This algorithm is a random string matching algorithm, which can find out the position where the pattern string appears for the first time in the text string. The basic idea of this algorithm is to calculate a corresponding hash value from a pattern string of length m according to a certain function. This function is called a hash function (or a fingerprint function). Each substring of length m in the text string is also calculated according to this function to obtain a corresponding hash value. Only those text substrings that have the same hash value as the pattern string are likely to

match the pattern string. If the hash value of the text substring is not the same as the hash value of the pattern string, then the text substring must not match the pattern string.

5.1.1 KR String Matching Algorithm Description

The KR string matching algorithm is described as follows: Assume a set $\sum$, the size of the set as S, the string can be regarded as an S-ary number, that is to say, when the pattern string $P = p_1p_2p_3...p_m$, each character(char) can be selected Inner code (such as ASCII code), then P can be numericalized as:

$$p = p_1 \times s^{m-1} + p_2 \times s^{m-2} + ... + p_m \tag{4}$$

At this point, a larger prime number q can be found, and the hash function of p is defined as the numerical result of the P string modulo a larger prime number q:

$$\phi(p) = mod(p, q) = mod(p_1 \times s^{m-1} + p_2 \times s^{m-2} + ... + p_m, q) \tag{5}$$

In the same way, for text strings, the above operations can also be performed on a segment whose length is m. If the text string is assumed to be $T = t_1t_2t_3...t_n$, an integer $k(k < n - m + 1)$ can be found in it, and T among them $T(k) = t_{k+1}, t_{k+2}, ..., t_{k+m}$, the numerical value of the S-ary for this segment is:

$$T'(k) = t_{k+1} \times s^{m-1} + t_{k+2} \times s^{m-2} + ... + t_{k+m} \tag{6}$$

Then the hash value of $T'(k)$ is:

$$\phi(T'(k)) = mod(T'(k), q) = mod(t_{k+1} \times s^{m-1} + t_{k+2} \times s^{m-2} + ... + t_{k+m}, q) \tag{7}$$

Then compare $\varphi(T'(k))$ with $\varphi(p)$, if not equal, keep comparing $\varphi(T'(k+1))$ until you find an equal value or a hash value of all text substrings is compared. If the hash value of the corresponding (some) text substring is equal to the pattern string, each corresponding element needs to be compared to ensure that the result is correct.

For example: The elements in the pattern string P and the text string T are taken from the set $\sum = \{0, 1, 2, ..., 9\}$, then $S = 10$. For the text string $T =$ "23590231415267399 21", the pattern string $P =$ "31415". Then according to formula (1), the pattern string P can be numericalized as:

$$p' = p_1 \times 10^{m-1} + p_2 \times 10^{m-2} + ... + p_m = 3 \times 10^4 + 1 \times 10^3 + ... + 5 = 31415$$

The substring $T(k)$ of length m starting from the kth element of the text string T can be numericalized according to formula (3) as:

$$T'(k) = t_{k+1} \times 10^{m-1} + t_{k+2} \times 10^{m-2} + ... + t_{k+m} (k < n - m + 1)$$

When $k = 0$, then:

$$T'(0) = 2 \times 10^4 + 3 \times 10^3 + 5 \times 10^2 + 9 \times 10 + 0 = 23590$$

When $k = k + 1$, then:

$$T'(k+1) = t_{k+2} \times 10^{m-1} + t_{k+3} \times 10^{m-2} + ... + t_{k+m+1} = T'(k) \times 10 - t_{k+1} \times 10^{m} + ... + t_{k+m+}$$

So given $T'(k)$, the value of $T'(k+1)$ can be calculated in linear time (if the constant S^{m-1} has been calculated before calculating $T'(k+1)$)

Through $T'(0)$ can get $T'(1) = 10(23590 - 2000) + 2 = 35902$

The following uses a fingerprint function to calculate the hash value:

Let q be a prime number randomly selected in the interval $[1, nm^2]$, to select the fingerprint function as $\phi(p) = mod(p, q)$, that is, the hash value of the pattern string or a substring equal to the length of the pattern string: After the string is digitized. The remainder obtained by dividing the value of by a randomly chosen prime number.

Assuming that q is a prime number 13, then $\phi(P) = mod(31415, 13) = 7$. The hash value obtained by using the above hash function for each substring of the text string T with a length of 5 is as follows:

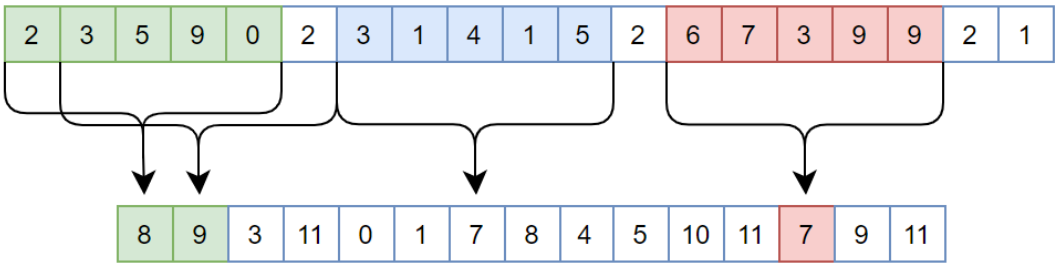

Figure 5: Example of KR algorithm

Through the above process, it can be seen that the fingerprints

of the substring with a length of 5 starting from the 7th position in the text string and the fingerprint of the substring with a length of 5 starting from the 13th position are the same as the fingerprints of the pattern string, both of which are 7. But only the substring in the 7th position matches the pattern string.

5.1.2 Time Complexity Analysis of KR String Matching Algorithm

According to the above description of the KR string matching algorithm, the KR algorithm can be divided into the following two steps:

- First calculate the hash value of the $n - m + 1$ substrings in the text string and the pattern string.
- Perform a matching search for each element of the string with the same hash value, and return the position of the corresponding substring in the text string if all corresponding elements match.

First, analyze the time complexity of the first step: for the process of calculating the fingerprint of the pattern string, $m - 1$ times of multiplication and m times of addition are required, and the required time is $\Theta(m)$. The time required to compute the first substring $T(0)$ of the text string T is still $\Theta(m)$. Because

$$T'(k+1) = T'(K) \times 10 - t_{k+1} \times 10^m + t_{k+m+1} = 10(T'(k) - t_{k+1} \times 10^{m-1}) + t_{t+m+1} (k < n-$$

If 10^{m-1} is pre-calculated before calculating $T(k+1)$, it can be seen from the above expression that calculating $T(k+1)$ from $T(k)$

only takes linear time, then calculating all the lengths of the T string. The time required for a substring of m is $\Theta(n-m)$. And the calculation of 10^{m-1} can be realized in $\Theta(m)$ time.

The second step can be seen as matching two strings of the same length. If the match is successful (that is, each corresponding element is equal), the starting position of the substring is returned. If there is a mismatch (that is, two elements are not equal), -1 is returned, indicating that the two strings do not match. In fact, this process is the comparison process of strings. The worst case of this comparison process is that each substring is compared with the pattern string once, then the number of comparisons required is $(n-m+1)m$ times, and the time required in the worst case is $\Theta((n-m+1)m)$.

5.1.3 KR Algorithm Related Technologies

The key to the KR string matching algorithm is to convert a string into an integer, which is also called the hash code of the string. In the conversion process, it is particularly important that each hash code should minimize the occurrence of collisions, that is, the situation where different strings are converted into the same hash code.

Although a string can be regarded as composed of multiple chars, the same char set will form different strings due to different combinations of chars. Such as "*step*" and "*pets*". If the sum of the integers corresponding to each char is simply used as the hash code, all strings composed of the same set of chars will conflict with each other. Therefore, the process of calculating the hash code for

the string adopts the numerical process introduced in Section 1.1.1. This method is also called the method of polynomial hash code, that is, a non-zero constant $a \neq 1$ is taken. For the string "$x_0x_1...x_{n-1}$", its hash code can be taken as: $x_0a^{n-1} + x_1a^{n-2} + ... + x_{n-2}a^1 + x_{n-1}$. From a mathematical point of view, this is equivalent to using the chars in the string as the coefficients of each item in a polynomial in turn, so it is called "polynomial hash code".

In addition, for the choice of prime number q. Choosing an appropriate q plays a key role in reducing conflicts. If q is selected to be very small, different hash codes will have the same result after dividing q, that is, the probability of obtaining the same hash value is relatively high. This situation is called a conflict. If q is too large, the purpose of compressing the hash code will not be achieved. At present, there is no standardized formula to calculate the value of the prime number q, but the prime number q generally takes a random number smaller than M, and the value of M can be calculated by the following formula: $M = \frac{1}{prob\{collision\}}$(the denominator is the allowable probability of conflict).

5.2 Realization of RKR-GST Algorithm

Since the GST algorithm compares each element of the text string and the pattern string one by one, if any unequal elements are found, the text string will start comparison from the beginning. Therefore, the number of comparisons is relatively large. The RKR-GST algorithm is improved on the basis of the GST algorithm, and the KR algorithm with higher efficiency is introduced, so the algorithm does not need to compare each element in the text string

and the pattern string. Only when the hash value of the substring of the pattern string is the same as the hash value of the substring of the text string, the comparison is carried out. This method improves the operating efficiency of the GST algorithm to a large extent.

5.2.1 RKR-GST algorithm extension to KR algorithm

The KR algorithm is a pattern matching algorithm, and it is impossible to determine which string is a pattern and which string is text when comparing the similarity of two strings, because the two strings may contain some similarities, and the two strings It may be very long, so it is not appropriate to simply use one of the strings as a pattern string to calculate the similarity with the KR algorithm. The RKR-GST algorithm extends the KR algorithm to meet the needs of comparing the similarity of two strings. The expansion mainly includes the following three aspects:

- Take one of the strings as a pattern string and divide it into several substrings of length s, such as $p_p, p_{p+1}, ..., p_{p+s-1}$, and then calculate the hash of each substring through a certain fingerprint function. Use the same method for text strings to find the hash values of all substrings with length s.

- Each hash value in the pattern string is compared with all hash values in the text string; if two hash values are found to be equal, there may be two substrings in the pattern string and the text string that both match. But it does not mean that the found match is a maximum match, because the length

of the match at this time is s, and there may be matches longer than this length. Therefore, it is necessary to further expand the matching until the matching fails, that is, one of the following three possibilities occurs: the elements in the two strings do not match, an element in the two strings has been marked. And one of the two strings, the string has been matched. If the matching is successful, that is, the matching length is greater than or equal to s, the obtained matching substring is added to the maximal-matches set.

- The KR algorithm ends when the comparison between the hash code of the last substring of the text string and the hash code of the pattern string is completed, and the condition for the end of the RKR-GST algorithm is that no longer matching substring longer than the search length s can be found end of string.

5.2.2 RKR-GST algorithm description

Because the GST algorithm uses the method of comparing each element in the two strings one by one in the process of searching for the maximum match, if it finds a mismatch, it will go back. This approach is inefficient and has room for improvement. The RKR-GST algorithm is an improved algorithm based on the GST algorithm. The RKR-GST algorithm uses the more efficient KR algorithm in the process of searching for the maximum match, and the ultimate goal is to find all the maximum matches that exist in the two text strings. The basic idea of this process is the same as that of the GST algorithm. Find the maximum match and then

add a mark to the maximum match found, and then enter the next round of search until there is no match that meets the requirements.

Definition 1: Searching for the minimum matching length (represented by the parameter s) in the loop is called the search length.

The highest-level pseudocode of the implementation process of the RKR-GST algorithm is shown below:

Algorithm 1 Algorithm highest level pseudocode

```
searchLengths ← initialSearchLength
stop ← false
repeat
    /* L_max is the size of the largest match found in this loop  */
    L_max ← scanpattern(s)
    if L_max > 2 × s then
        /* Quite a long string, don't mark it as "tile", try it with
           a larger s                                                */
        s ← L_max
    else
        /* Generate "tile" from the largest set of matches found   */
        markstrings(s)
          if s > 2 × minimumMatchLength then
            s ← s div 2
          else if s > minimumMatchLength then
            s > minimumMatchLength
          else stop ← true
until stop;
```

It can be seen from the pseudo-code above that the RKR-GST algorithm is very similar to the GST algorithm. The main difference is that the GST algorithm starts from the longest match and finds all the matches from high to low. Every time a match is found, Just

mark the corresponding positions of the T string and the P string. The RKR-GST algorithm divides the pattern string P into several substrings of the same length, and calculates their hash values, and then compares these hash values with the hash table established by each substring of the corresponding length of each text string T, and further match in the case of equality, try to find the longest possible match, and then add the found match to the queue that stores the largest match. It can be seen from this process that each time a certain search length is searched, the matches added to the maximum matching queue may be multiple and may overlap, so the RKR-GST algorithm is required to have a program segment that can handle the maximum matching queue. This process is implemented by the function *markstrings*(), which is the function of *Marking*. The implementation process of RKR-GST algorithm is described as follows:

First set an initial value for the search length s. Next is a loop process to find all the largest matches that meet the matching conditions. This loop ends when the search length s reaches the minimum matching length. In this cycle mainly includes two functions: scanpattern and markstrings. The main operation of the scanpattern function is to find all the largest matches that occur in this cycle, and return a value, which is used as the new search length to search for the largest match in the next round. The main operation of the markstrings function is to mark the corresponding position in the text string and the pattern string of the largest match found during the execution of the scanpattern function. When this process is completed, the value of s is updated, if its value is less than the minimum matching length given by the algorithm in advance, the

algorithm ends, otherwise the algorithm will use the new s value as the search length for the next round of maximum matching search.

6 Implementation details

7 Evaluation

8 Results

The following results were achieved in this work.

- Conducted the survey of existing plagiarism detection tools and techniques.
- Proposed the solution for quick interactive code plagiarism detection and code quality check.
- Designed and implement plagiarism detection algorithm.

References

[1] AIKEN A. MOSS : A system for detecting software plagiarism // http://www.cs.berkeley.edu/⊠aiken/moss.html. — 2004. — Access mode: https://cir.nii.ac.jp/crid/1570572700250505984.

[2] Baker B.S. On finding duplication and near-duplication in large software systems // Proceedings of 2nd Working Conference on Reverse Engineering. — 1995. — P. 86–95.

[3] Baker Brenda S. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance // SIAM Journal on Computing. — 1997. — Vol. 26, no. 5. — P. 1343–1362. — https://doi.org/10.1137/S0097539793246707.

[4] Baxter Ira D. DMS: Program Transformations for Practical Scalable Software Evolution // Proceedings of the International Workshop on Principles of Software Evolution. — New York, NY, USA : Association for Computing Machinery. — 2002. — IWPSE '02. — P. 48–51. — Access mode: https://doi.org/10.1145/512035.512047.

[5] Li Zhenmin, Lu Shan, Myagmar Suvda, and Zhou Yuanyuan. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. // OSdi. — 2004. — Vol. 4. — P. 289–302.

[6] Baxter I.D., Yahin A., Moura L., Sant'Anna M., and Bier L. Clone detection using abstract syntax trees // Proceedings.

International Conference on Software Maintenance (Cat. No. 98CB36272). — 1998. — P. 368–377.

[7] Prechelt Lutz, Malpohl Guido, Philippsen Michael, et al. Finding plagiarisms among a set of programs with JPlag. // J. Univers. Comput. Sci. — 2002. — Vol. 8, no. 11. — P. 1016. — Access mode: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=b7909f36e772cc99216e36dc2e4e0919c81ec1fe.

[8] Gitchell David and Tran Nicholas. Sim: A Utility for Detecting Similarity in Computer Programs // The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education. — New York, NY, USA : Association for Computing Machinery. — 1999. — SIGCSE '99. — P. 266–270. — Access mode: https://doi.org/10.1145/299649.299783.

[9] Halstead Maurice H. Elements of Software Science (Operating and programming systems series). — Elsevier Science Inc., 1977.

[10] Kamiya T., Kusumoto S., and Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code // IEEE Transactions on Software Engineering. — 2002. — Vol. 28, no. 7. — P. 654–670.

[11] Komondoor Raghavan and Horwitz Susan. Using Slicing to Identify Duplication in Source Code // Static Analysis / ed. by Cousot Patrick. — Berlin, Heidelberg : Springer Berlin Heidelberg. — 2001. — P. 40–56.

[12] Sallis P., Aakjaer A., and MacDonell S. Software forensics: old methods for a new science // Proceedings 1996 International Conference Software Engineering: Education and Practice. — 1996. — P. 481–485.

[13] Schleimer Saul, Wilkerson Daniel S., and Aiken Alex. Winnowing: Local Algorithms for Document Fingerprinting // Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. — New York, NY, USA : Association for Computing Machinery. — 2003. — SIGMOD '03. — P. 76–85. — Access mode: https://doi.org/10.1145/872757.872770.

[14] Verco K. L. and Wise M. J. Plagiarism à la Mode: A Comparison of Automated Systems for Detecting Suspected Plagiarism // The Computer Journal. — 1996. — 01. — Vol. 39, no. 9. — P. 741–750. — https://academic.oup.com/comjnl/article-pdf/39/9/741/993714/390741.pdf.

[15] Weiser Mark. Program Slicing // IEEE Transactions on Software Engineering. — 1984. — Vol. SE-10, no. 4. — P. 352–357.

[16] Wise Michael J. YAP3: Improved Detection of Similarities in Computer Program and Other Texts // SIGCSE Bull. — 1996. — mar. — Vol. 28, no. 1. — P. 130–134. — Access mode: https://doi.org/10.1145/236462.236525.