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 технологической 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 plagiarism. First of all, we must find students who plagiarize. 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 objects 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 Forensic 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 on this project 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 tag 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 tag 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 tag series 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, 16] and YAP [21] 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.

1 Problem statement

The goal of this work is to implement a toolkit for student source code plagiarism detection based on code search.

- Conduct the survey of existing techniques, tools and algorithms used for plagiarism detection.
- Design a solution for quick interactive code plagiarism detection.
- Develop a lightweight code search engine.
- Implement the plagiarism detection algorithm.
- Conduct an experimental study on a toolkit.

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, 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 Behavior. From the 1980s to the present, researchers at home and abroad 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 contained in the program code, maps these attributes to the vector space, and then calculates the similarity between the two. The software scientific measurement method [10] 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 softwares. Most of the subsequent 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 [15]. considered capacity, control flow, data dependence, nesting depth, and control structure six parts of program 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" [18]. 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 pro-

gram code into a program dependency graph (PDG) [14], and then used program slicing technology (Program Slicing) [19] 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 [9], YAP3 system, CCFinder system [13], 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) [9] 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:

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

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 [21] 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 [1, 16] 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 [1, 16]. 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 [7] 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.

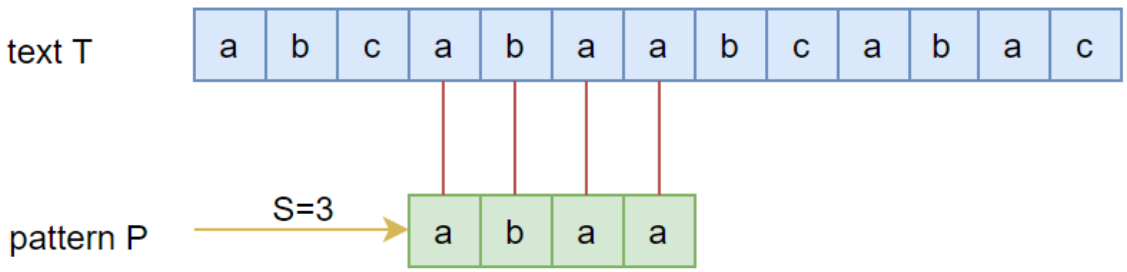

Figure 1: String pattern matching

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.

2.3 Local search engine

2.3.1 Description of Whoosh

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.

2.3.2 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:

```
import os.path
from whoosh.index import create_in
from whoosh.index import open_dir

if not os.path.exists('index'):
    os.mkdir('index')
ix = create_in('index', schema)
ix = open_dir('index')
```

3 Architecture

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

3.1 Input module

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

3.2 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.

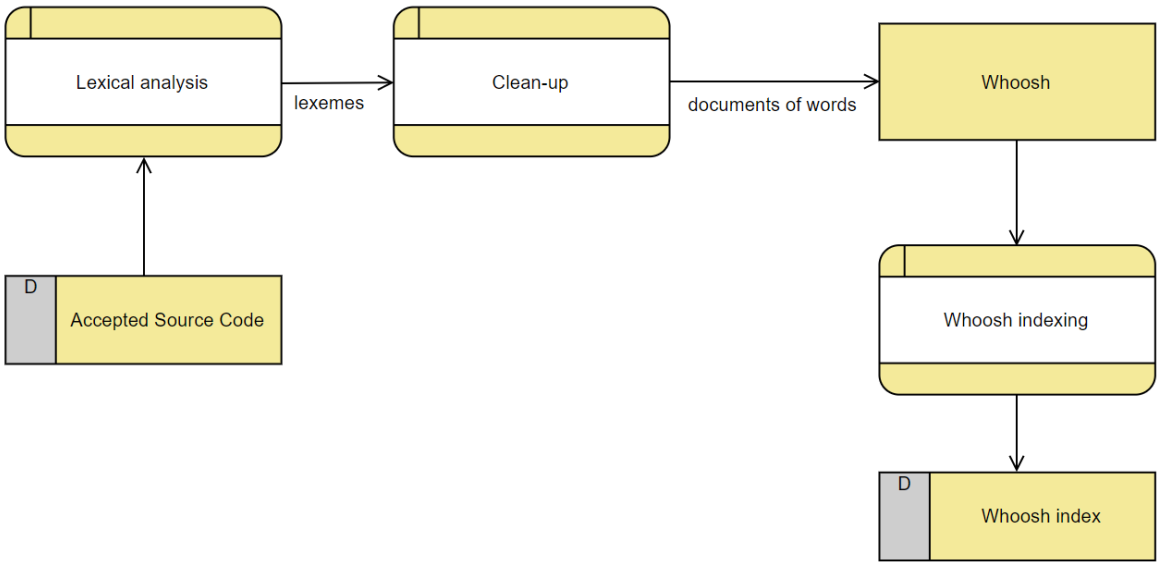

Figure 2: Workflow of indexing

3.3 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.

3.4 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.

3.5 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.

3.6 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.

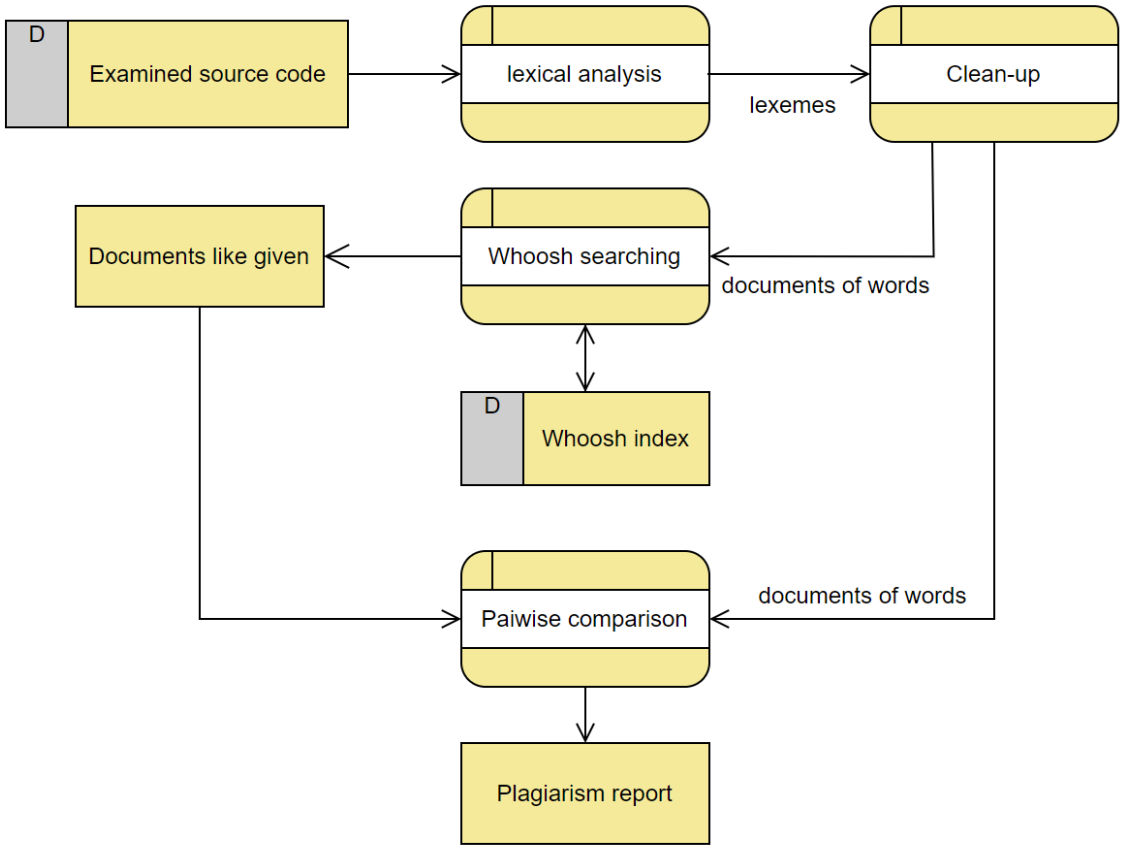

Figure 3: Workflow in search

3.7 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).

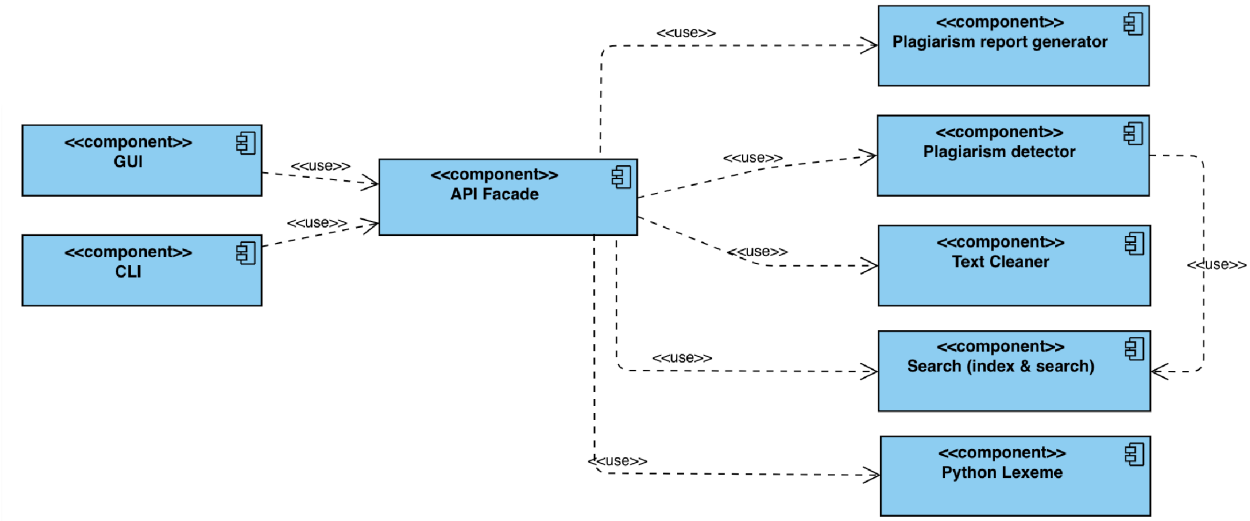

Figure 4: Main components

4 Lightweight code search engine based on Whoosh

4.1 Indexing process

4.1.1 Inverted Index

The key technology of Whoosh is to build an inverted index. The inverted index records which documents contain a certain word. For example, the word "LOVE" appears in the article you are reading. Suppose the number of this article is 111. , then a record of LOVE: 111 will be recorded in the index. When you search for the word "LOVE", the search engine finds the document corresponding to LOVE from the inverted index. If there are more than one, the search engine calculates the correlation between the document and the search term, and sorts it according to the correlation and returns it to You end up.

4.1.2 Word segmentation

When we search, the keyword we search for may be a sentence, and there are so many contents in the document, but the index only records the mapping relationship between the word and the document number. Participate. For source code, word segmentation is a simple matter, because the code is composed of several words.

4.1.3 Index Mode

Now to build an index for the source code, we need to define an index for meaningful strings in the source code, and then store it.

4.2 Search process

In the search process, you need to use the $open_dir$ function to open the index file and create a Searcher object.

5 Plagiarism detection algorithm

Because the plagiarism of the program code may change the writing position of variables, functions, etc., after the program code is converted into a tag string through grammatical analysis, the position order of these description information in the tag string may be different, so for this change of order case, the string matching algorithm should be able to find such a match. In addition, if a code segment in a program has been matched with another code segment, it cannot participate in the matching of other code segments again. This is done to prevent inaccurate matching results caused by repeated writing of code segments with the same function. Case. Also, long similar snippets should have a higher chance of matching than shorter similar snippets. This is done so that long similar program segments are more likely to be plagiarized than short ones, so string matching algorithms should be able to handle this.

From the definition of similarity, it can be seen that the string matching algorithm applied in the program code plagiarism detection system should be able to find all matching pairs in the two string sequences, so as to conveniently calculate the value of the similarity of the two program codes. Not only that, the string matching algorithm can also find the position of the same subsequence of the two sequences, and then mark similar code segments, providing a basis for teachers to judge plagiarism.

A string matching algorithm suitable for a program code plagiarism detection system usually has the following four important characteristics [12].

(1) Each match is one-to-one correspondence. That is to say, suppose a substring in a text string A repeats many times, but there is only one matching substring in another text string B. Then the substring that appears first in A matches the corresponding substring in B string successfully, and other repeated substrings in A will be skipped. This feature indicates that repetition of a certain portion of code in the source text will be ignored by the algorithm. At the same time, it is guaranteed that a matching substring in a string is not recalculated multiple times.

(2) Where the matching substring of the two texts is located in the text

has no or little influence on finding the match. This feature indicates that adjusting the order of some code segments in the source program will not affect the matching result of the final comparison algorithm. For example, in a *switch* statement, the order in which *case* statements are written will not affect the matching result.

(3) Randomly inserting or deleting some tokens (token) has a small impact on the calculation results of the similarity. For example, dividing a long sentence into several shorter sentences expresses the same meaning. The calculated results should not significantly reduce the similarity.

(4) Long string matching should be prioritized over short strings. This is because longer similar code segments reflect a greater likelihood of plagiarism.

5.1 Existing solution

5.1.1 Levenshtein Algorithm

The Levenshtein distance (LD) algorithm [8] was designed by Russian scientist Vladimir Levenshtein in 1965 to measure (measure) the similarity between two strings, assuming that the two strings s are source strings (source), and t (target) is target string. The LD distance is the number of steps required for deletion, insertion or replacement operations from the source string s to the target string t. For example:

$t =$ "$test$", $s =$ "$test$" then $LD(s, t) = 0$

$t =$ "$test$", $s =$ "$tent$" then $LD(s, t) = 1$

The larger the LD, the greater the difference between the source string and the target string.

The algorithm is used in spell checking, speech recognition, DNA analysis, and plagiarism detection. The detection result of this algorithm is the same as that of the LCS algorithm, and it can only achieve effective search for sequence matching, so it will not be able to detect plagiarism that changes the writing order when applied to a plagiarism detection system.

5.1.2 Dynamic Programming Algorithm

In the SIM system and the YAP system, a dynamic programming algorithm [9] is used to compare two tagged strings. This algorithm also belongs to the LCS algorithm and is used to solve the LCS problem. The implementation process of the algorithm can be described as follows: the strings to be matched are first arranged by inserting spaces between the characters of the strings, so that a series of permutations are obtained, and then the strings obtained by the two programs are combined Permutations are compared to get the largest match. For example: *masters* and *stars* are two character strings with different lengths, and the lengths of the two character strings can be made the same by inserting spaces. You can insert spaces in the *stars* string to make it equal to the length of the *masters* string. This can have many different permutations.

Such as:

masters or *masters*

star rs *stars*

Use m to represent the matching value, d to represent the non-matching value, and g to represent the distance between the two. A *block* is the character sequence corresponding to the maximum matching value in the array at this time. For example: $m = 1, d = -1$ then $g = -2$. For these two permutations, rs/rs and $sters/stars$ are the largest *block* sequences. The corresponding values are $2, 3$. Find the largest value from all *blocks*, which is the largest matching substring. The search method adopts the method of dynamic programming (dynamic programming).

The method is described as follows:

For two strings s and t, it is necessary to determine the arrangement value D(i,j) between $s[1...i]$ and $t[1...j]$, that is,

$$max_{1\leq i\leq|s|,1\leq j\leq|t|}D(i,j)thenD(i,j)=max\begin{cases}D(i-1,j-i)+score(s[i],t[j])\\D(i-1,j)+g\\D(i,j-i)+g\\0\end{cases}\tag{4}$$

in

$$score(s[i],t[j])=\begin{cases}m, & if s[i]=t[j]\\ d & others\end{cases}\quad D(1,i)=i\times g, D(j,1)=j\times g\tag{5}$$

The execution result of the algorithm for Example 1 is the same as that of the LCS algorithm.

5.1.3 Heckel Algorithm

This algorithm is a matching algorithm described by Paul Heckel [11]in the literature for comparing two versions of a source program or file to reveal all the differences in them. This algorithm is used in the YAP2 system. In this algorithm, a line is used as the detection unit, a line is a continuous unit in two texts, a line appears only once in each text, and each corresponding element is required to be equal, but the line is in the text The location in can be different. If several such permutations are found and they are adjacent to each other, combine them into one permutation. This process is repeated until adjacent permutations are found. In this way, several long-length permutations (longest substrings) are found, and their total length will be used to calculate the similarity. The algorithm is realized through the following two processes: one is to find a permutation; the other is to expand the permutation to make it as long as possible. The time complexity of this algorithm is linear. For Example 1, the result obtained after passing the Heckel algorithm is as follows 5:

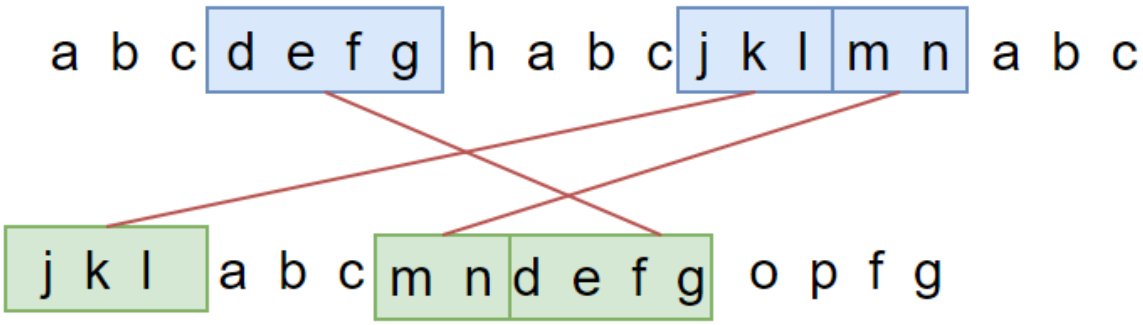

Figure 5: Heckel Algorithm.

5.2 Algorithm implemented on the tool

LCS (Longest Common Subsequences, referred to as LCS algorithm) [20] is used to solve the LCS problem. The description of the LCS problem is as follows: delete zero or more characters from the given two strings to obtain the longest Long sequences of identical characters. LCS is the longest common sequence of the two strings. In addition, no substring longer than it can be found in the two strings. Of course, the LCS may not be unique. Because, two strings can have multiple maximum common sequences of the same length.

Example 1: The two strings are $string1 =$ "$abcdefghabcjklmnabc$" and $string2 =$ "$jklabcmndefgopfg$", then the LCS matching result is shown in Figure 7:

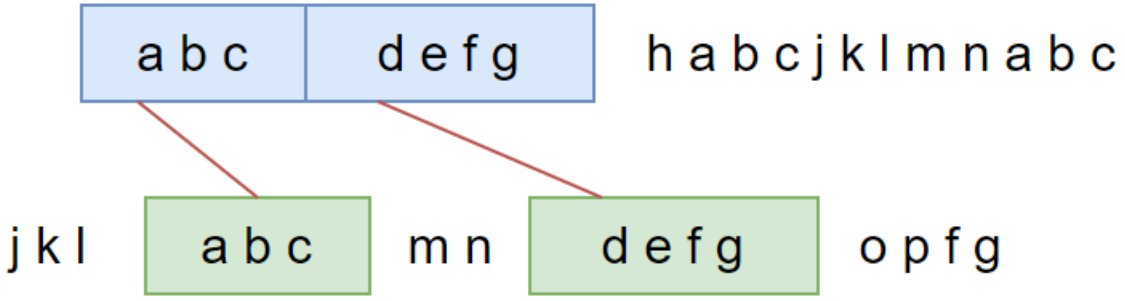

Figure 6: LCS algorithm.

The LCS algorithm has a wide range of applications, and the initial research on the LCS problem is to study it as a differential compression algorithm. For example, in a version management system, a file is continuously modified to produce different versions, and the new version does not change much compared to the old version. In order to save storage space, we can compare the old file version with the modified new version to find out what parts are the same and what parts are different. This eliminates the need to store both the original and modified files separately, but only the old file version and the different parts of the old and new versions.

The comparison of two file versions is similar to the LCS problem. Due to the high time complexity of the LCS algorithm, it has basically withdrawn from this application. However, in the version management system, when comparing and merging different versions of the same file, the LCS algorithm still plays an important role.

The LCS algorithm can also be used in the field of genetic engineering. For example, the basic method to determine a disease-causing gene is to compare the DNA strands of patients with the disease with those of healthy individuals, find out the same parts and different parts, and then analyze them.

It has been proved that the time complexity of the LCS algorithm cannot be less than $O(mlog(n))$ [17].

The LCS algorithm is ordered, that is to say, the obtained common substrings are all strictly ordered, which makes it useless for the detection of sample strings whose writing order is changed.

5.2.1 LCS Algorithm

LCS (Longest Common Subsequences, referred to as LCS algorithm) [20] is used to solve the LCS problem. The description of the LCS problem is as follows: delete zero or more characters from the given two strings to obtain the longest Long sequences of identical characters. LCS is the longest common sequence of the two strings. In addition, no substring longer than it can be found in the two strings. Of course, the LCS may not be unique. Because, two strings can have multiple maximum common sequences of the

same length.

Example 1: The two strings are $string1 =$ "$abcdefghabcjklmnabc$" and $string2 =$ "$jklabcmndefgopfg$", then the LCS matching result is shown in Figure 7:

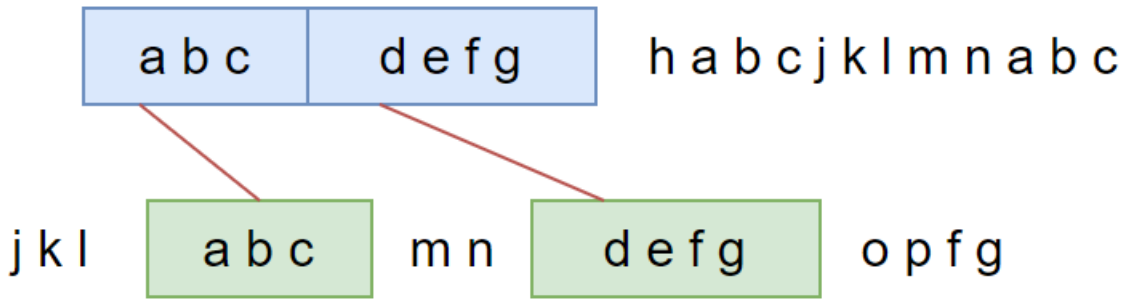

Figure 7: LCS algorithm.

The LCS algorithm has a wide range of applications, and the initial research on the LCS problem is to study it as a differential compression algorithm. For example, in a version management system, a file is continuously modified to produce different versions, and the new version does not change much compared to the old version. In order to save storage space, we can compare the old file version with the modified new version to find out what parts are the same and what parts are different. This eliminates the need to store both the original and modified files separately, but only the old file version and the different parts of the old and new versions.

The comparison of two file versions is similar to the LCS problem. Due to the high time complexity of the LCS algorithm, it has basically withdrawn from this application. However, in the version management system, when comparing and merging different versions of the same file, the LCS algorithm still plays an important role.

The LCS algorithm can also be used in the field of genetic engineering. For example, the basic method to determine a disease-causing gene is to compare the DNA strands of patients with the disease with those of healthy individuals, find out the same parts and different parts, and then analyze them.

It has been proved that the time complexity of the LCS algorithm cannot be less than $O(mlog(n))$ [17].

The following is the pseudo-code implementation of the algorithm in the program:

```
Data: S_1, S_2: strings of lexemes; m: minimal length of copied text
Result: F: What fraction of S_1 is common between S_1 and S_2
common_lexemes ← 0
found_lcs ← true
while found_lcs do
    found_lcs ← false
    s_1, s_2 ← Ratcliff_Obershelp(S_1, S_2)
    /* s_1 and s_2 are lists of intervals of S_1 and S_2        */
    for (s'_1, s'_2) ∈ zip(s_1, s_2) do
        if |s'_1| ≥ m then
            found_lcs ← true
            fill S_1[s'_1] with symbol α
            fill S_2[s'_2] with symbol β
            /* now S_1[s'_1] ≠ S_2[s'_2]: text fragments get different,
               consisting of α and β respectively            */
            common_lexemes ← common_lexemes +|s'_1|
        end
    end
end
F ← common_lexemes/|S_1|
```

Figure 8: Implementation of longest common subsequence

6 Results

The following results were achieved in this work.

- Conducted the survey of existing plagiarism detection tools and techniques.
- Conducted the survey of existing solutions.
- Developing the search engine based on Whoosh.
- Implemented LCS algorithm on the plagiarism detection.

The following tasks must be done to complete this work.

- Implementing the Plagiarism Detection Algorithm in the Whoosh-based Search Engine.
- Further improvements to the String Matching algorithm
- Expanding to more programming languages

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] Gilleland Michael et al. Levenshtein distance, in three flavors // Merriam Park Software: http://www. merriampark. com/ld. htm. — 2009.

[9] 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.

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

[11] Heckel Paul. A Technique for Isolating Differences between Files // Commun. ACM. — 1978. — apr. — Vol. 21, no. 4. — P. 264–268. — Access mode: https://doi.org/10.1145/359460.359467.

[12] Hoffmann M. The Plagiarism Detector Copy-D-TEC // Department of Computer Science, University of Stellenbosch, South Africa, Tech. Rep. Final Project Report. — 2004.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Ullman J. D., Aho A. V., and Hirschberg D. S. Bounds on the Complexity of the Longest Common Subsequence Problem // J. ACM. — 1976. — jan. — Vol. 23, no. 1. — P. 1–12. — Access mode: https://doi.org/10.1145/321921.321922.

[18] 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.

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

[20] Whale Geoff. Plague: plagiarism detection using program structure. — School of Electrical Engineering and Computer Science, University of New ..., 1988.

[21] 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.