Journal of Computing and Communication Vol.2 , No.2 , PP. 8-18 , 2023

Automated Detection Approaches for Source Code Plagiarism in Students' Submissions

Mohamed Ashraf, Abdelrahman Sayed, Shereen E. Elbohy, Essam Eliwa*

Department of Computer Science, Misr International University, Cairo, Egypt

*Corresponding Author: Essam Eliwa [***essam.eliwa@miyegypt.edu.eg***]

ARTICLE DATA

Article history:
Received 23 May 2023
Revised 28 June 2023
Accepted 01 July 2023
Available online

Keywords:
Plagiarism detection, automatic assessment, programming education.

ABSTRACT

Code plagiarism is a significant concern in software development, as it compromises the integrity of original work and can lead to ethical and legal consequences. The need for effective plagiarism detection techniques has grown in parallel with the rise in online coding resources and collaborative platforms. The paper analyses existing plagiarism detection tools, comparing their characteristics, functions, and development timelines. Emphasis is placed on essential factors such as additional case detection, direct display of matched pairings, and compatibility with multiple programming languages. By examining these features, educators and software developers can decide which tools best suit their needs.

Additionally, the paper explores various plagiarism detection techniques, including attribute counting, content comparison, string tiling, and parse tree comparison. The advantages and limitations of each method are examined, underscoring the need for continuous improvement and innovation in the field. This paper presents the most widely available plagiarism detection tools that can be seamlessly integrated into learning management systems. In conclusion, the paper highlights critical areas for future research and development in plagiarism detection. These include the integration of plagiarism detection with live learning management systems to streamline the process for educators and students, the enhancement of usability and user experience in plagiarism detection tools to facilitate their adoption, the advancement of detection algorithms to improve accuracy, and the support for multi-language and cross-language comparisons to cater to diverse programming environments.

1. Introduction

Plagiarism in source code refers to presenting someone else's code as one's own, intentionally or unintentionally, without proper attribution [1]. In educational settings, detecting code plagiarism manually among numerous student submissions is a daunting task for instructors in terms of feasibility and cost-effectiveness. Consequently, developing software technologies to assist educators in identifying code plagiarism has become necessary. However, designing software that can reliably detect plagiarism poses challenges due to the intricate nature of the problem.

Existing tools for code plagiarism detection often require manual intervention to determine whether the similarities detected between programs or modules indicate plagiarism or arise from legitimate reasons, such as using standard implementations for a given assignment. Instances such as students drawing inspiration from examples provided in class, collaborating on assignments, or mere coincidences can lead to apparent similarities. Therefore, even when suspicion arises, teachers usually prefer to hear the student's perspective before making a conclusive decision or discussing the matter with others.

In code plagiarism cases, plagiarised software is usually created by minor alterations to an existing program, often involving simple text replacements that do not require an in-depth understanding of the code. Integrating code plagiarism detection tools with Learning Management Systems offers numerous benefits for both educators and students. It streamlines the detection process, facilitates timely feedback and intervention, reduces the administrative burden on instructors, and promotes a culture of academic integrity and responsible coding practices among students. By embracing this modern approach, educational institutions can establish an environment that nurtures originality, critical thinking, and ethical conduct in software development and programming education. Detecting code plagiarism

is essential for upholding the integrity of original work, ensuring fair assessment, and promoting ethical coding practices. Traditional plagiarism detection methods rely heavily on textual similarity measures, such as string-matching algorithms or token-based comparisons. While these approaches can identify exact matches or partial similarities in code, they often fall short when faced with modifications like variable renaming, code rearrangement, or syntactic changes.

This paper aims to explore methodologies and systems specifically designed to assist instructors in detecting potential instances of code plagiarism. It will highlight the distinctive characteristics of each system and provide performance data to empower educators in making informed decisions about which tool to employ for plagiarism detection. Several applications are available for this purpose, including offline systems like Sherlock [2], YAP [3, 4], Plaggie [5], SIM [6, 7], Marble [8], and CPD [9]. Online tools such as JPlag [6, 10] and MOSS [11, 12], as well as tool packages like CodeSuite [13], are also available to aid instructors in their endeavours. By exploring state of the art in code plagiarism detection, this paper aims to assist educators in effectively addressing code plagiarism, promoting academic integrity, and fostering an environment cultivating originality and responsible coding practices among students.

2. Research Objectives and Method

We review the various implementation approaches used in similarity detection technologies. It will discuss the tools identified in the literature and on the Internet, evaluating whether they should be included in our comparison. Additionally, the study will examine several comparative studies [14–17] comparable to ours. Various plagiarism techniques have been presented to identify source code written in C, C++, or Java [18, 19]. Each method focuses on different aspects of code plagiarism. Some methods are primarily used to compare source codes written in various programming languages.

In contrast, others are designed to handle complex code modifications but may take longer to detect than popular ones. This section presents a variety of projects for detecting plagiarism and source code. Methods of determining source code similarity are classified as structural, semantic, attribute-based, or behavioural for this study. Each of these measures can be used to compare and assess the similarity between two sections of source code.

3. Literature Review

In this section, we provide an overview of the techniques used by plagiarism detection tools, discuss research objectives and methodologies, and delve into the different approaches and algorithm designs employed in plagiarism detection software.

3.1. Attributes-Based Technique

The attribute-based technique detects plagiarism by counting attributes in the program code. This method offers distinct advantages, such as performance speed and efficient handling of many source codes [20]. It involves selecting and highlighting properties in the program code and assigning each one a distinct token. The proportion of similarity is then estimated by counting the distinct tokens that match between program codes.

Early methods relied heavily on counting attributes, representing each file with a numerical sequence [21]. Two files were identified as suspected examples of plagiarism if their sequences were "close enough." However, this approach proved ineffective for concise programs and non-trivial program changes, leading to a shift towards attribute-counting and control-flow-based techniques.

3.2. Structure-Based Technique.

Systems were developed that compare program structures rather than translate them into numerical sequences. The structure-based method, which uses tokenisation and string-matching algorithms to determine similarity, is acceptable for detecting plagiarism in programming courses. Structural similarity can be evaluated using textual strings, employing techniques like string edit distance and string alignment to compare source code. Existing plagiarism detectors like Plague, YAP, and JPlag utilise structure-based approaches. Software plagiarism detection systems commonly employ either attribute-counting methods or structure-based approaches. A careful study shows that structure-based techniques are more successful than

attribute counting. Based on the structure, software dependency networks can also be evaluated, as used in [22].

3.3. Semantic-Based Technique.

Semantic similarity of source code refers to the similarity in meaning. This is accomplished through semantic analysis, which examines source code to extract information not expressed in a computer language's syntax. Semantic techniques typically analyse a program using program dependency graphs (PDGs), which graphically represent a procedure's data and control dependencies. Tools like JPlag use this method to identify plagiarism by mining program dependency graphs.

3.4. Behavioral-Based Technique.

Behavioural similarity, also known as dynamic analysis, identifies similar runtime behaviours of source code. Various strategies are used to find behavioural similarities, including evaluating functional equivalence, recognising similar interactions with the execution environment, or identifying similar program logic.

3.5. Machine Learning-Based Technique.

Machine learning-based techniques have emerged as practical approaches for automated source code plagiarism detection. This section discusses two primary methods commonly used in the field.

1) *Supervised Learning Approach:* The supervised learning approach involves training a model on labelled data, where each submission is categorised into different levels of plagiarism. The process begins with semantic feature extraction preprocessing on the source code and other factors such as comment similarity, submitted file length, and output console strings. The program then ranks the top 100 feature pairings and incorporates them into the dataset. For the supervised learning approach, there are two common scenarios:
   - Binary Classification: In this scenario, the goal is to classify submissions into plagiarised or non-plagiarised categories. The model is trained to differentiate between these two classes.
   - Multi-level Categorisation: This scenario involves categorising submissions into multiple levels of plagiarism, which may include partial copy cases. The model is trained to classify submissions into different levels, such as no copy, partial copy, or complete copy.

   One example of the supervised learning approach is presented in [23], where a Support Vector Machine (SVM) model was trained after categorising the data. However, alternative classifiers such as deep learning models like Recurrent Neural Networks (RNN) with Long Short-Term Memory (LSTM) support can also be utilised for this purpose.

2) *Unsupervised Learning Approach:*
   The unsupervised learning approach aims to group source code submissions based on similarity without using labelled data. Before applying any clustering or grouping algorithms, feature extraction preprocessing on the source code is performed. The submissions are then grouped based on the differences in distance between their extracted features. The unsupervised learning approach can be helpful in scenarios where labelled data is scarce or difficult to obtain. It allows for identifying potential plagiarism cases based on patterns and similarities within the source code. This approach has been explored in [24], where submissions were clustered based on the differences in feature distances. Both supervised and unsupervised learning approaches provide valuable techniques for detecting source code plagiarism using machine learning.
   These approaches leverage the power of algorithms and models to analyse large datasets and identify patterns indicative of plagiarism. Duracilk et al. [25] presented another system using a new scalable approach by using an incremental clustering approach to achieve modularity and scalability of the

algorithm. The process involves converting the source code into an abstract syntax tree, which is then used to generate characteristic vectors. These vectors are clustered using the incremental k-means algorithm and stored in a database. Real student submissions were used to test this approach, and results were compared to MOSS [26] system. The results show that the proposed approach outperforms the MOSS system regarding precision and recall. Specifically, the proposed approach achieved a precision of 0.95 and a recall of 0.91, while the MOSS system achieved a precision of 0.89 and a recall of 0.81.

Other notable examples exist of using unsupervised learning techniques for automatic plagiarism detection in programming assignments. One such example is presented in [24], where a novel hybrid approach is proposed. This approach leverages static features extracted from the intermediate representation of a program within a compiler infrastructure, such as GCC. The system uses these extracted features to create models of the programming assignments. Unsupervised learning techniques are then applied to these feature representations to detect similarities and patterns indicative of plagiarism. Using unsupervised learning in this context allows for identifying potential plagiarism cases without relying on labelled data. By examining the similarities between the feature representations, the system can detect instances where programming assignments exhibit highly similar patterns, suggesting possible plagiarism.

3.6. Design-Based Technique.

The design-based technique is a unique approach for detecting source code plagiarism that focuses on analysing design patterns within the code. Design patterns are recurring solutions to common programming problems that embody best practices and provide a structured approach to software development. In the design-based technique, the detection process involves retrieving the design patterns used in the source code and determining the similarity percentage based on these patterns [27].

By comparing the design patterns in different code submissions, it becomes possible to identify instances of plagiarism where similar design patterns are used. The advantage of the design-based technique is its ability to detect plagiarism even when the code has been modified or altered to some extent. It goes beyond simple text comparison and delves into the underlying structure and organisation of the code. This approach is particularly effective in detecting cases where students may have slightly modified the code logic while keeping the overall design pattern intact. The technique can identify similarities in the arrangement of classes, methods, and relationships between different code components by analysing the design patterns. However, it is essential to note that the design-based approach requires a comprehensive database of known design patterns to compare against. This database is a reference for identifying similarities and determining the similarity percentage. To implement the design-based technique, specialised tools and algorithms have been developed that analyse the structural aspects of the code and extract the relevant design patterns [28].

These tools automate the process and provide efficient means of identifying potential plagiarism based on the design patterns present in the source code.

3.7. Style-Based Technique.

The style-based technique aims to identify the author of the source code by analysing and capturing their unique coding style features. This method utilises individual authors' distinct characteristics and patterns to determine whether a contribution is plagiarised [29]. After collecting all student submissions, a preprocessing step is performed to prepare the files for analysis. The following steps are typically involved:

a) Removal of unnecessary whitespace: Any extraneous white space, such as extra indentation or empty lines, is eliminated. This step helps standardise the formatting across the source code files.
b) Deletion of file headers: The headers at the beginning of each source code file, which often contain information like file name, author, and creation date, are removed. This step ensures that the analysis focuses solely on the coding style rather than any metadata associated with the file.
c) Calculation of file size: The size of each source code file is determined by counting the number of lines present. This metric provides a primary measure of the code's complexity and length.

d) Counting loop structures: The occurrence of common loop structures, such as "for," "if," "if-else," and "while," is counted within each source code file. These loop terms have been identified as indicative of coding style and can help estimate the complexity of the code.
e) Categorisation of coding style features: An algorithm categorises the features into predetermined categories based on the collected information. This categorisation facilitates the comparison and identification of similarities or disparities between the coding styles of different authors.

4. Tools in Comparison

In this section, we introduce the tools selected for our comparison. The tools are presented in alphabetical order.

JPlag [30] was created by Guido Malpohl at the University of Karlsruhe. JPlag is an open-source web-based tool for plagiarism detection. It started as a student research project in 1996 and later became a web service called JPlag in 2005 by Emeric Kwemou and Moritz Kroll. JPlag adopts a structure-based approach by converting programs into token strings representing their structure. For comparing these token strings, JPlag utilises Michael Wise's "Greedy String Tiling" approach [3] with additional improvements for enhanced performance. It supports programming languages like Python3, Java, C-Sharp, C, C++, Scheme, and plain text. JPlag's results are presented through a robust graphical interface, generating HTML pages that allow for detailed exploration and explanation of observed similarities. The tool works in two stages: (1) parsing or scanning the comparison source codes to extract lexical tokens and (2) comparing these token strings using the Running- Karp-Rabin greedy string tiling technique to measure program similarity. JPlag provides clear result listings and an editor for comparing contributions, making it highly user-friendly. It also offers a Java Web Start client for easy file uploading to the server.

Marble [8] is an open-source tool created by the first author in 2002 at Utrecht University. It aims to detect questionable resemblances among Java submissions simply and easily. Marble facilitates the addition of new languages by utilising code normalisers to create language-independent tokens. It employs the RKS-GST method to find comparable tokens. Marble follows a structure-based technique for comparing contributions. It splits each contribution into sections, with each section containing only one top-level class. The tool performs normalisation by removing elements that students can easily change, such as comments, whitespace, and string constants. It also abstracts tokens to their token type and retains frequently used keywords and class/method names. Marble supports languages like Java, C, Perl, PHP, and XSLT. The tool outputs the results in a script format, providing a similarity score, the length of files, and the differences for each pair that surpasses a specified similarity threshold. Marble can be operated locally or remotely, allowing for comparison with earlier versions' files.

SCoSS (Source Code Similarity) was created at the Hanoi University of Science and Technology as a tool specifically designed for detecting plagiarism in competitive programming competitions. SCoSS employs various algorithms and metrics to detect plagiarism in multiple files. It is a web service primarily intended for non-commercial purposes. However, an open-source CLI version of the program is available for use. SCoSS supports languages like C++, Python, Java, and PHP. The tool utilises measures like SMOSS, which relies on the MOSS API. The results are outputted as a similarity array between each input file in the CLI version. The similarities are presented as percentages based on selected criteria such as operators and keywords. SCoSS allows the exclusion of files based on size and performs detailed plagiarism analysis when the similarity value exceeds the threshold. It is a Python CLI program available under the MIT License, and a web interface has also been developed for SCoSS.

MOSS [11,12] (Measure Of Software Similarity) is an abbreviation for "Measure Of Software Similarity." It was created by Aiken and colleagues at Stanford University in 1994 [26]. MOSS is an open-source web-based service that allows users to compare source code written in various programming languages, including C, C++, Java, Pascal, Python, Ada, ML, Lisp, Scheme, JavaScript, FORTRAN, Visual Basic, Haskell, Modula2, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, and HCL2.

Table 1 Key Open Source Plagiarism detection tools with their characteristics, supported languages, and approaches.

Tool	**MOSS**	**JPlag**	**Marble**	**SCoSS**
Open source	Yes	Yes	Yes	Yes
Local/online	Web-based	Web-based	Local	Web-based
Supported language	22	6	1	4
Developed year		1996	2002	2021
Developer	A. Aiken S. Schleimer D. Wilkerson	Guido Malpohl	Jurriaan Hage	Martin Borchert
Approach	Winnowing	Greedy String Tiling	Flax Lexical analyser	Counts and checks operators then compare the hash

One of MOSS's key features is its utilisation of the document fingerprinting method called "Winnowing." By dividing the code into k-grams, or continuous substrings, MOSS performs document fingerprinting to determine the similarity between standardised versions of code submissions. Users can specify the value of k, tailoring the analysis to their specific requirements. Upon conducting the comparison, MOSS generates an intuitive HTML display with clickable links and an embedded HTML diff editor. This user-friendly interface simplifies navigation through the comparison reports, allowing users to identify similarities and differences in the code easily. The reports are served as web pages on the MOSS server, facilitating convenient access and sharing.

MOSS also can automatically filter out matches to shared code, such as libraries or code provided by the instructor. This feature reduces the chances of false positives arising from direct code sharing. Additionally, MOSS allows users to indicate which submissions are recent and which are older from the command line interface. By default, MOSS compares files based on inputs or folders but also provides an option for file-to-file comparison within the submitted code.

Plaggie [5] was designed in 2002, by Ahtiainen et al. from the Helsinki University of Technology. It is a stand-alone source code plagiarism detection engine under the GNU General Public License for evaluating Java programming exercises Plaggie. In terms of design and functionality, it is similar to JPlag; however, Plaggie has several features that distinguish it from JPlag; in contrast to JPlag, which was originally offered as an online service, Plaggie is a software that must be installed locally and whose source code is accessible so it may be shared online or as a web service. It's a soul Java command-line application. Tokenisation is the basic approach for comparing source code in Plaggie, followed by Greedy String Tiling [30] and no optimisation. Plaggie had some drawbacks, including:

- Adapting to new languages is difficult and time-consuming.
- The feedbacks are not instantly clear.
- Performance issues and requires a lot of extra UNIX applications. This causes issues with portability.

YAP In 1992 [3, 31], Michael Wise designed the first version (YAP1), which was quickly followed by a more efficient version (YAP2). Wise released the latest edition of YAP (YAP3) in 1996, which could cope with transposed sequences well. YAP, which stands for 'Yet Another Plague,' is a plagiarism detection system that has been regularly upgraded since its beginnings in Australia in 1996. The Plague plagiarism detection program is the foundation for the YAP line of tools. YAP3 is the most recent version. While the tokenisers mentioned in [31] have been enhanced, they are essentially the same from YAP3 [32] ahead, with the difference that the tokens are now integers instead of strings.

SIM [34] SIM, according to Dick Grune, can detect plagiarism in software projects, detect possibly duplicated code fragments in huge software projects, and is resistant to typical alterations like name changes, rearranging statements and functions, and adding/deleting comments and white spaces. Like the Sherlock software, SIM is an efficient open-source plagiarism detection tool that runs on a single machine locally and includes a command-line interface. SIM's similarity approach begins with tokenising the source code, then creating a forward reference table that may be used to discover the best matches between new files and the text, and finally comparing both [35]. Programs are initially parsed using the flex lexical analyser to create a series of numbers, or tokens, in this method. The tokens for symbols, keywords, and comments, as well as the tokens for identifiers, are dynamically issued and saved in a common symbol table, with whitespace removed. The second program's token stream is divided into parts, each representing a software module, and each section is aligned with the first program's token stream independently. SIM evaluates programs' accuracy, style, and originality to find similarities. The main drawback was the lack of a graphical user interface.

5. Implementation Strategies and Proposed Integration Approach

This section provides a concise overview of various comparison strategies reported in the literature. Many techniques in these tools begin by converting the source code into a token string. Tokens, lexical items such as identifiers and keywords, serve as the program's building blocks. Along the way, source code elements that can be easily modified without affecting the program's meaning, such as comments, variable names, and indentation, are removed. The resulting token representations of the program are then examined for commonalities and similarities.

Similarity detection strategies are a type of plagiarism detection technology [36]. The once-common attribute counting approaches are an excellent illustration. Attribute counting algorithms (such as [37] and [38]) provide unique "fingerprints" for collection files, which include data like average line length, file size, and average number of commas per line. Files with identical fingerprints are processed in the same way. Small fingerprint records may clearly be compared quickly, but this method is currently regarded as inaccurate and is rarely utilised [39]. Modern plagiarism detection systems frequently include content comparison techniques. Among the most often used algorithms are string tiling, calculating joint coverage for a pair of files, and parse tree comparison [40, 41]. The comparison method should be called for each potential file pair identified in the input collection because these strategies usually work for file pairs. Also, the Fast Plagiarism Detection Technique (FPDS) [42] aims to increase the algorithmic efficiency of plagiarism detection by storing input collection files in a specific indexed data structure.

Document Fingerprinting: One commonly used strategy for plagiarism detection is based on N-Grams [43]. This approach converts each string or sentence into numerical representations to amplify their differences. It is crucial to implement the logic of this algorithm efficiently to avoid any data loss. Additionally, each document has its position occurrence of each code block, emphasising the importance of carefully selecting the source code sections to be compared [26].

```
def generate_ngrams(s, n):
    s = s.lower()

    s = re.sub(r'[^a-zA-Z0-9\s]', ' ', s)

    tokens = [token for token in s.split(" ") if token != ""]

    ngrams = zip(*[token[i:] for i in range(n)])
    return [" ".join(ngram) for ngram in ngrams]
```

Figure 1 Code for generating N-Grams

Tokenisation [44] is widely used in computer systems to prevent renaming variables and altering loop types. Simple tokenisation methods replace program code parts with single tokens. All identifiers, for example, can be

replaced with' IDT' tokens and all values with' VALUE' tokens. As a result, a = b + 45; will be changed with' IDT'='IDT'+'VALUE'; As a result, renaming variables hasn't benefited the plagiarist [45].

These techniques play a vital role in identifying similarities and instances of plagiarism in various documents and source codes. By utilising efficient and effective algorithms, plagiarism detection tools can accurately detect and highlight instances of plagiarism.

5.1 Key Considerations for Effective Tools

Several factors should be taken into consideration to ensure the effectiveness of plagiarism detection tools:

- **Additional Cases Detection**: Effective tools should be able to detect additional cases beyond the basic ones, such as do-while conditionals, to provide comprehensive results.
- **Direct Display of Matched Pairings**: Tools that directly display the matched pairings on the source codes simplify the analysis and enable a better understanding of the similarities found.
- **Exclusion of Base Code**: The examination should exclude the base code from comparison to focus solely on identifying similarities among the submitted codes.
- **Detailed HTML Reports**: Tools should generate HTML reports that provide detailed findings, facilitating the analysis and interpretation of the results.
- **Flexible Comparison Weights**: It is beneficial for tools to allow modifications to the weights used in the comparisons, enabling users to customise the importance assigned to different code elements.
- **Language Compatibility:** A good tool should be compatible with multiple programming languages, accommodating many submissions.
- **Graphical User Interface (GUI):** The availability of a graphical user interface (GUI) enhances the user experience and makes working with the tool more intuitive and efficient.
- **Threshold and Weight Experimentation:** Tools that allow experimentation with various scenarios, such as adjusting the threshold and weights, empower users to fine-tune the tool's settings and optimise its performance for their specific needs.

5.2 Proposed Integration Approach

Plagiarism detection needs to be easily integrated with the learning management system. This section presents a proposed method for integrating plagiarism detection with LMS using the JPlag tool. JPlag adopts a structure-based approach by converting programs into token strings that represent the code's underlying structure. To compare these token strings, JPlag utilises Michael Wise's "Greedy String Tiling" [4] approach, incorporating additional improvements for enhanced performance. JPlag supports several programming languages, including Python3, Java, C-Sharp, C, C++, Scheme, and plain text. The results obtained from JPlag are presented through a robust graphical interface. The tool takes a collection of programs as input, compares them pair by pair, and generates a series of HTML pages that facilitate in-depth exploration and explanation of the observed similarities. The operation of JPlag occurs in two stages.

First, all source codes for comparison are parsed or scanned, depending on the input language, and transformed into a stream of lexical tokens. Then, using the Running-Karp-Rabin greedy string tiling technique, the extracted token strings are compared pairwise to determine the similarity between each pair of programs. Program similarity is quantified by measuring the proportion of tokens from one program that can be overlaid onto another during each comparison. The proposed method leverages the JPlag algorithm, which applies a structure-based approach to plagiarism detection. By converting programs into token strings and utilising advanced comparison techniques, JPlag enables the accurate identification of similarities among programs written in different languages. The tool's graphical interface further enhances the analysis and interpretation of the detected similarities.

An abstract representation of code snippets is created, where each code element is assigned a unique identifier. This representation enables the comparison of code structures across different programming languages and coding styles, eliminating the need for language-specific parsers. By comparing the pairwise structural similarities between code snippets, potential cases of code plagiarism can be identified. Unlike traditional text-based similarity measures, the Paired Structural Metric offers a more fine-grained code structure analysis. It can detect similarities even when the plagiarised code has undergone modifications

such as variable renaming, code rearrangement, or syntactic changes. This makes it a robust and accurate approach for identifying instances of code plagiarism. Focusing on the structural aspects of code enhances the effectiveness of plagiarism detection techniques and helps maintain the integrity of software development. So, in this study, JPlag compares the hash value of the pattern to the hash value of the current sub-string one by one, and if they match, the process proceeds to match individual characters. Hashing, in a nutshell, is the act of transforming a string to a numerical value to increase efficiency. As a result, if two strings are equivalent, their hash values are also identical. The hashed pattern is compared to the beginning of the string, and the window size is set to the length of the pattern, as shown in Figure 2.

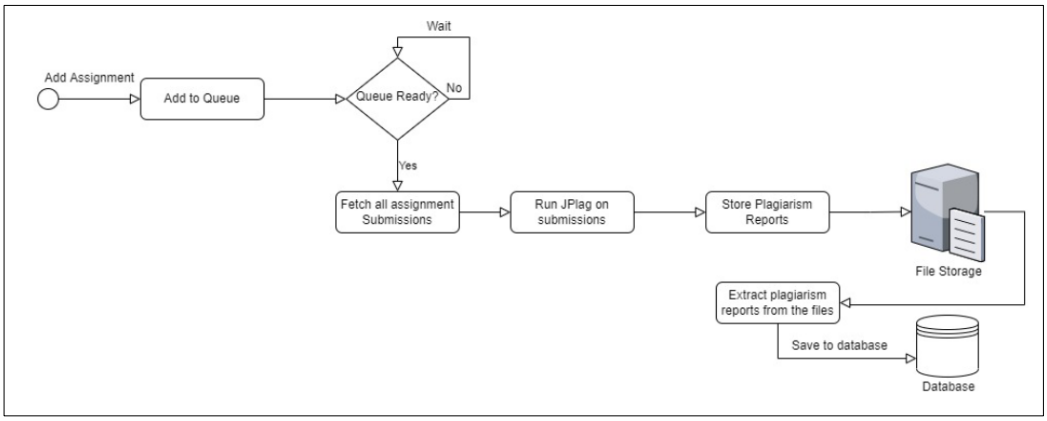

Figure 2 Proposed Integration Model for Plagiarism Checking.

6. Conclusion

In this paper, we explored the field of plagiarism detection and examined various techniques and algorithms used in this domain. We focused on understanding the principles behind plagiarism detection tools rather than delving into the implementation details. By reviewing the literature, we gained insights into different comparison strategies and their effectiveness in identifying similarities and instances of plagiarism. We discussed attribute counting approaches, which generate unique fingerprints for files based on attributes such as line length, file size, and the number of commas per line. However, these methods are considered inaccurate and are rarely used in modern plagiarism detection systems. Instead, content comparison techniques have gained prominence. Algorithms like string tiling, joint coverage calculation, and parse tree comparison have proven effective in comparing file pairs and detecting similarities. We also examined the Fast Plagiarism Detection Technique (FPDS), which aims to improve the efficiency of plagiarism detection by utilising indexed data structures for storing input collection files. This technique enables faster and more efficient detection of instances of plagiarism.

Additionally, we explored specific algorithms employed in plagiarism detection, such as document fingerprinting based on N-Grams. This algorithm converts strings or sentences into numerical representations, enhancing their differentiation. We emphasised the importance of carefully selecting the source code sections to be compared, considering the unique position occurrence of each code block within documents. Furthermore, tokenisation emerged as a widely used approach to prevent variable renaming and manipulation of loop types, making it more challenging for plagiarists to conceal their actions. Practical plagiarism detection tools should consider various factors, including additional case detection, direct display of matched pairings in source code, exclusion of base code from examination, generation of detailed HTML reports, flexibility in adjusting comparison weights, compatibility with multiple programming languages, and the inclusion of a user-friendly graphical interface. Furthermore, experimenting with different scenarios to determine optimal thresholds and weights can enhance the accuracy and reliability of plagiarism detection results. In conclusion, plagiarism detection is crucial in maintaining academic integrity and protecting intellectual property.

7. Future Work

In the future, we envision several areas of research and development to enhance plagiarism detection and its usability. These include:

- **Integration with Live Learning Management Systems:** We plan to focus on integrating plagiarism detection seamlessly with live learning management systems. By incorporating plagiarism detection tools directly into the existing workflow of instructors and educational platforms, we aim to streamline the process and make it more convenient for instructors to assess student work and promote academic integrity.
- **Improving Usability and User Experience:** enhancing usability and user experience is crucial. This involves designing intuitive interfaces that are easy to navigate, providing clear and actionable feedback to instructors and students, and integrating visualisation techniques to aid in result interpretation. By prioritising user-centred design principles, we can create more user-friendly, efficient, and practical tools to detect plagiarism.
- **Advancing Accuracy of Detection Algorithms:** Researchers should continue exploring ways to improve the accuracy of plagiarism detection algorithms further. This can be achieved by exploring novel similarity comparison techniques, refining existing algorithms, and developing hybrid approaches that combine multiple methods. By continuously pushing the boundaries of detection accuracy, we can ensure that plagiarism detection tools remain effective in identifying plagiarism, even as techniques for disguising similarities evolve.
- **Supporting Multi-language and Cross-language Comparisons:** Extending plagiarism detection to support multiple programming languages and enable cross-language comparisons is an essential area for future work.

By addressing these future directions, we can advance the field of plagiarism detection, making it more accessible, accurate, and adaptable to the evolving landscape of software development and education.

References

[1] Strickroth, Sven. "Plagiarism Detection Approaches for Simple Introductory Programming Assignments." In Proceedings of the Fifth Workshop" Automatische Bewertung von Programmieraufgaben"(ABP 2021), virtual event, October 28-29, 2021. 2021.

[2] Franca B Allyson et al. "Sherlock N-Overlap: invasive normalisation and overlap coefficient for the similarity analysis between source code". In: IEEE Transactions on Computers 68.5 (2018), pp. 740–751.

[3] Michael J Wise. "YAP3: Improved detection of similarities in computer program and other texts". In: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education. 1996, pp. 130–134.

[4] Xiao Li and Xiao Jing Zhong. "The source code plagiarism detection using AST". In: 2010 International Symposium on intelligence information processing and trusted computing. IEEE. 2010, pp. 406–408.

[5] Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. "Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises". In: Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006. 2006, pp. 141–142

[6] EnilPajic´andVedranLjubovic´. "Improving plagiarism detection using genetic algorithm". In: 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE. 2019, pp. 571–576.

[7] Ljubovic, Vedran, and Enil Pajic. "Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories." IEEE Access 8 (2020): 96505-96514.

[8] Ludlow, Barbara L. "Virtual reality: Emerging applications and future directions." Rural Special Education Quarterly 34, no. 3 (2015): 3-10.

[9] M Jiffriya, MA Jahan, and R Ragel. "Plagiarism detection tools and techniques: A comprehensive survey". In: Journal of Science-FAS-SEUSL 2.02 (2021), pp. 47–64.

[10] Anala A Pandit and Gaurav Toksha. "Review of plagiarism detection technique in source code". In: International Conference on Intelligent Computing and Smart Communication 2019. Springer. 2020, pp. 393–405.

[11] Dana Sheahen and David Joyner. "TAPS: A MOSS extension for detecting software plagiarism at scale". In: Proceedings of the Third (2016) ACM Conference on Learning@ Scale. 2016, pp. 285–288.

[12] Dirson Santos de Campos and Deller James Fer- reira. "Plagiarism detection based on blinded logi- cal test automation results and detection of textual similarity between source codes". In: 2020 IEEE Frontiers in Education Conference (FIE). IEEE. 2020, pp. 1–9.

[13] Ilana Shay, Nikolaus Baer, and Robert Zeidman. "Measuring whitespace patterns as an indication of plagiarism". In: (2010).

[14] Thomas Lancaster and Fintan Culwin. "A comparison of source code plagiarism detection engines". In: Computer Science Education 14.2 (2004), pp. 101–112.

[15] Jurriaan Hage, Peter Rademaker, and Nike` Van Vugt. "A comparison of plagiarism detection tools". In: Utrecht University. Utrecht, The Netherlands 28.1 (2010).

[16] Mayank Agrawal and Dilip Kumar Sharma. "A state of art on source code plagiarism detection". In: 2016 2nd International Conference 135. on Next Generation Computing Technologies (NGCT). IEEE. 2016, pp. 236–241.

[17] Marwah Najm Mansoor and Mohammed SH Al- Tamimi. "Computer-based plagiarism detection techniques: A comparative study". In: International Journal of Nonlinear Analysis and Applications 13.1 (2022), pp. 3599–3611.

[18] Arwin, Christian, and Seyed MM Tahaghoghi. "Plagiarism detection across programming languages." In Proceedings of the 29th Australasian Computer Science Conference-Volume 48, pp. 277-286. 2006.
[19] Franclinton, Ricardo, Oscar Karnalim, and Mewati Ayub. "A Scalable Code Similarity Detection with Online Architecture and Focused Comparison for Maintaining Academic Integrity in Programming." (2020): 40-52.
[20] S.K. Robinson J.A. Faidhi. "An empirical approach for detecting program similarity and plagiarism within a university programming environment." In: Comput. Educ. 11(1). 1987, pp. 11–19.
[21] Edward L Jones. "Metrics based plagiarism monitoring". In: Journal of Computing Sciences in Colleges 16.4 (2001), pp. 253–261.
[22] Chao Liu et al. "GPLAG: detection of software plagiarism by program dependence graph analysis". In: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. 2006, pp. 872–881.
[23] B. KattaJ J. Yasawi. "Plagiarism detection in programming assignments using deep features". In: 4th Asian Conference on Pattern Recognition. 2017, pp. 7–54.
[24] B. KattaJ J. Yasawi. "Unsupervised learning based approach for plagiarism detection in programming assignments". In: Machine Learning for Source- code Plagiarism Detection. 2017, pp. 9–71.
[25] Michal Dˇurac ́ık, Emil Krsˇa ́k, and Patrik Hrku ́t. "Scalable Source Code Plagiarism Detection Using Source Code Vectors Clustering". In: 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS). 2018, pp. 499–502. DOI: 10. 1109 / ICSESS. 2018. 8663708.
[26] A. Aiken S. Schleimer D. Wilkerson. "Winnowing: local algorithms for document fingerprinting". In: European Conference on Technology Enhanced Learning. ACM. 2003, pp. 76–85.
[27] M. Basavaraju A. Asadullah. "Design patterns based preprocessing of source code for plagiarism detection". In: 2012 19th Asia-Pacific Software Engineering Conference vol. 2012, pp. 128–13
[28] Jing Dong, Yajing Zhao, and Yongtao Sun. "A Matrix-Based Approach to Recovering Design Patterns". In: IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans 39.6 (2009), pp. 1271–1282. DOI: 10.1109 / TSMCA.2009.2028012.
[29] G. Cosma O.M. Mirza M. Joy. "Style analysis for source code plagiarism detection—an analysis of a dataset of student coursework". In: IEEE 17th International Conference on Advanced. 2017, pp. 296–297.
[30] Prechelt, L., Malpohl, G., & Philippsen, M. (2002). Finding plagiarisms among a set of programs with JPlag. J. Univers. Comput. Sci., 8(11), 1016.
[31] Michael J Wise. "Detection of Similarities in Student Programs: YAP'ing may be Preferable to Plague'ing". In: Acm Sigcse Bulletin 24.1 (1992), pp. 268–271.
[32] Georgina Cosma and Mike Joy. "An approach to source-code plagiarism detection and investigation using latent semantic analysis". In: IEEE transactions on computers 61.3 (2011), pp. 379–394.
[33] Paul Heckel. "A technique for isolating differences between files". In: Communications of theACM 21.4 (1978), pp. 264–268.
[34] Tapan P Gondaliya, Hiren D Joshi, and Hardik Joshi. "Source Code Plagiarism Detection' SCPDet': A Review". In: International Journal of Computer Applications 105.17 (2014).
[35] Divya Luke et al. "Software plagiarism detection techniques: A comparative study". In: (2014).
[36] Richard M Karp and Michael O Rabin. "Efficient randomised pattern-matching algorithms". In: IBM Journal of research and development 31.2 (2011), pp. 249–260.
[37] Ste ́phane Ducasse, Matthias Rieger, and Serge Demeyer. "A language independent approach for detecting duplicated code". In: Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).' Software Maintenance for Business Change' (Cat. No. 99CB36360). IEEE. 1999, pp. 109–118.
[38] Jinan AW Faidhi and Stuart K Robinson. "An empirical approach for detecting program similarity and plagiarism within a university programming environment". In: Computers & Education 11.1 (2000), pp. 11–19.
[39] AS Bin-Habtoor and MA Zaher. "A survey on plagiarism detection systems". In: International Journal of Computer Theory and Engineering 4.2 (2012), p. 185.
[40] David Gitchell and Nicholas Tran. "Sim: a utility for detecting similarity in computer programs". In: ACM Sigcse Bulletin 31.1 (2008), pp. 266–270.
[41] Boumediene Belkhouche, Anastasia Nix, and Johnette Hassell. "Plagiarism detection in software designs". In: Proceedings of the 42nd annual Southeast regional conference. 2010, pp. 207–211.
[42] Maxim Mozgovoy et al. "Fast plagiarism detection system". In: International Symposium on String Processing and Information Retrieval. Springer. 2005, pp. 267–270.
[43] Gaurav Toksha Anala A. Pandit. "Review of Plagiarism Detection Technique in Source Code". In: International Conference on Intelligent Computing and Smart Communication. 2020, pp. 393– 405.
[44] Mike Joy and Michael Luck. "Plagiarism in programming assignments". In: IEEE Transactions on education 42.2 (2012), pp. 129–133.
[45] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: A multilinguistic token-based code clone detection system for large scale source code". In: IEEE transactions on software engineering 28.7 (2006), pp. 654–670.