applied sciences

MDPI

Article

Review of Code Similarity and Plagiarism Detection Research Studies

Gunwoo Lee [1], Jindae Kim [2], Myung-seok Choi [1], Rae-Young Jang [1] and Ryong Lee [1,*]

[1] AI Data Research Center, Division of Science and Technology Digital Convergence, Korea Institute of Science and Technology Information (KISTI), Daejeon 34141, Republic of Korea; gwlee@kisti.re.kr (G.L.); raezero@kisti.re.kr (R.-Y.J.)
[2] Department of Computer Science and Engineering, Seoul National University of Science and Technology, Seoul 01811, Republic of Korea; jindae.kim@seoultech.ac.kr
* Correspondence: ryonglee@kisti.re.kr

Abstract: The foundational technique of code similarity detection, which underpins plagiarism detection tools, has already reached a level of maturity where it can be effectively employed for practical applications, demonstrating commendable performance. However, although the understanding of code clones—referred to as similar codes—has evolved, there has been a noticeable decline in the emergence of novel proposals for code similarity detection techniques. The landscape of code similarity detection techniques is diverse and can be divided based on how codes are represented. Each method, designed to cater to different types of detectable code similarity instances, has distinct advantages and drawbacks. Therefore, the selection of an appropriate method is crucial and is contingent on the specific objectives of the analysis. This paper provides a comprehensive exploration of code similarity detection techniques and illuminates the prevailing trends in plagiarism detection research. It acquaints readers with a spectrum of distinct code similarity detection methods, accompanied by the requisite contextual background knowledge. Additionally, it presents a detailed overview of the trajectory of research trends in plagiarism detection.

Keywords: code similarity detection; plagiarism detection; research trends

Citation: Lee, G.; Kim, J.; Choi, M.-s.; Jang, R.-Y.; Lee, R. Review of Code Similarity and Plagiarism Detection Research Studies. *Appl. Sci.* **2023**, *13*, 11358. https://doi.org/10.3390/app132011358

Academic Editor: Andrea Omicini

Received: 29 August 2023
Revised: 26 September 2023
Accepted: 11 October 2023
Published: 16 October 2023

1. Introduction

Modern software development occurs in an environment characterized by knowledge–sharing and collaboration. New technologies and ideas spread rapidly among developers, offering significant advantages in terms of innovation and efficiency. However, within this context, issues related to code similarity arise, requiring careful consideration [1,2]. Encouraged by an environment and culture that promote the exchange of information and mutual learning, developers often leverage code snippets to solve similar problems. However, these code fragments can exhibit structural and logical resemblances, potentially leading to problems associated with code duplication [3,4]. The duplication of code increases maintenance costs and raises the likelihood of bugs and security vulnerabilities. Furthermore, there is a darker side to knowledge sharing: the unauthorized copying or plagiarism of software code can breach copyright laws and lead to academic misconduct [5]. This dual problem of potentially infringing upon the rights of the original code authors, while undermining ethical standards within the development ecosystem, must be addressed [6].

Computer and programming education plays a crucial role in advancing the skills of developers, offering essential hands-on coding exercises and assignments. However, these code-based tasks, which are administered and performed using digital files, have become susceptible to easy replication, compounded by the vast availability of educational resources on the Internet [7,8]. This has led to growing concerns about plagiarism. Moreover, the global shift toward remote education due to the COVID-19 pandemic has resulted in a significant increase in online submissions for various assignments. Consequently,

the use of appropriate techniques and tools to detect plagiarism within these submissions has gained paramount importance. Thus, tackling code plagiarism has become a critical challenge within the education sector [9]. It not only underscores the significance of maintaining academic integrity but also emphasizes the need for robust plagiarism-detection mechanisms. Educational institutions are recognizing the importance of implementing effective strategies to curb plagiarism and promote ethical coding practices. These efforts are essential to fostering a fair learning environment and ensuring that the educational achievements of students are correctly evaluated and rightfully earned.

In this paper, we begin by providing the foundational knowledge required to comprehend the techniques employed to detect similar codes, which form the basis for plagiarism detection. These code similarity detection techniques can be categorized into text-based [10–15], token-based [16–21], tree-based [22–26], program dependence graph (PDG)-based [27–31], and hybrid methods [32–35], each determined by the approach used to represent the code structure. We then examine the evolution and current research directions of these techniques. Building upon this foundation, the primary objectives of this paper are twofold: first, to outline the ongoing research landscape of plagiarism detection techniques, highlighting the methodologies being explored; and second, to offer insights into key considerations and promising avenues for future research in this domain.

Based on the findings of our investigations, the process of plagiarism detection is composed of identifying potential instances of similar code and subsequently determining whether plagiarism exists within the identified similarities. Notably, those attempting plagiarism often employ diverse obfuscation strategies to avoid detection. Nonetheless, our exploration of code similarity detection techniques reveals that many of these obfuscation methods can be effectively countered. Given these insights, the trajectory of future research on plagiarism detection techniques may pivot toward developing decision support systems capable of assisting in determining plagiarism once suspected instances of similar code are identified. This avenue suggests a shift away from enhancing already effective code similarity detection techniques and toward crafting decision-making aids that can offer comprehensive support for addressing plagiarism-related concerns.

The remainder of this paper is organized as follows. Section 2 explains the background theory and terminology used in this report. Section 3 examines the trends in code similarity detection techniques. Section 4 provides additional information on research trends related to plagiarism detection and on available detection tools. Finally, Section 6 presents the conclusion and discusses future research directions.

2. Code Clones: Definition and Types

In this section, the concept of "code clone", which is a general term for similar code, is defined, and its types are explained. Additionally, background theories and terminologies related to code similarity and plagiarism detection techniques are presented to provide foundational knowledge for understanding the various techniques explained in later sections.

Generally, the term *code clones* refers to two identical code fragments. However, the adjective "same" is not a technical term, and sometimes similar code with some differences, rather than exact identicalness, is considered a code clone. Depending on how the similarity of two codes is defined, the method for detecting code clones may differ, and the meaning of the detected code clones may vary.

Here, we explain the definitions that distinguish code clones from types I to IV, in accordance with the methods by Davey et al. [36] and Bellon et al. [37], which are widely used to describe the types of code clones. Table 1 shows brief information about these different types.

Table 1. Types of code clones.

Classification	Types	Explanation
Text Similarity	Type I	Code snippets that are identical except for whitespace, comments, etc.
	Type II	In addition to type I, code snippets that are structurally or syntactically identical except for identifiers, literals, types, etc.
	Type III	In addition to type II, code snippets that include some structural/grammatical variations of additional statements or expressions but are nonetheless thought to be copied
Functional Similarity	Type IV	Code snippets that perform the same computation but differ in the actual structural/grammatical implementation

If the types of code clones are broadly classified, they can be grouped into type I, II, and III code clones, which are based on text similarity, and type IV code clones, which consider functional similarity.

Of all of the four types, type I is the case with the highest textual similarity, in which code parts are completely identical except for comments and spaces. In types II and III, the actual textual similarity becomes progressively lower. Finally, in type IV, even if there is no similarity based on text, functionally similar cases are judged and considered as code clones. In the following sections, each type of code clone is described in more detail, and examples and additional methods for classifying code clones besides types I–IV are introduced.

2.1. Type I Code Clone

Type I code clones, popularly known as "exact clones", consider only substantially identical pieces of code to be code clones. Note that whitespace characters or comments that are not involved in actual code execution are ignored in this comparison. Thus, type I code clones may still exhibit differences in the code, when examined using a general version control system (VCS), or in the text, when examined using the `diff` command in Unix systems. Table 2 demonstrates the similarities and differences that can exist between an original code and its type I code clone.

Table 2. Example of an original code snippet (left) and its type I code clone (right).

```
if (i >= j) {
  s = t + i; //comment 1
  t = i - j;
} else
  s = t + j; //comment 2
```

```
if(i>=j){
  s=t+i;
  //comment 1
  t=i-j;}
else //comment 2
  s=t+j;
```

In the second code snippet, most of the spaces between the keywords and variables have been removed, and the positions of comments `comment 1` and `comment 2` are completely different, even in their order relative to the actual code. Moreover, "`}`", which marks the end of the `if` statement, has been moved to the end of the line above it, before `else`. This is the result of moving the newline character "`\n`" in the line above to the end of "`}`". In the case of these two code snippets, the two codes will likely be assessed to be completely different by a simple text difference detection technique. However, in the context of code similarity detection, the second code snippet is considered a type I code clone, i.e., the strictest type, because there is no actual difference between it and the original, except for spaces and comments.

Ignoring code style differences, the type I code clone has exactly the same code. Notably, the identifiers or literals have been preserved. Thus, code similarity detection targeting type I code clones can be considered (1) when it is meaningful to compare variable names or function names, or (2) when the same code is detected among codes using the same literal. However, in such cases, even a slight change in variable names can cause similar codes to fail to be recognized as code clones.

2.2. Type II Code Clone

Code similarity detection targeting type II code clones is immune to certain transformations. In many cases, structurally or syntactically identical pieces of code use different identifiers or literals. Whereas the detection of type I code clones would assess such code as different, the detection of type II code clones ignores minor variations and focuses on the grammatical structure of code fragments to determine code clones. Thus, if the code similarity detection is based on the criteria for type II code clones, both code fragments can be detected as code clones.

Consider the following code snippet:

```
if (i >= j) {
  s = t + 1;
  t = i - j;
} else
  s = t + j;
```

Then, consider the next code snippet, which is a type II code clone of the previous code snippet:

```
if(a >= b) {
  c = d + 2;
  d = a - b;
}else
  c = d + b;
```

Although the code clone has many differences from the original code, and it is considered a type II code clone of the original code.

As with type I, differences in some whitespace characters are ignored. However, unlike with type I, the variables `i`, `j`, `s`, and `t` are changed to `a`, `b`, `c`, and `d`, respectively, and the integer literal `1` is changed to `2`. Although there may be differences in the actual calculation owing to the change in the integer literal, the structures of the two codes are the same, considering the assignment relationship of the variable names. What should be noted is that considering the grammatical structure of the code, the range of code that is determined to be a code clone may vary depending on how much variation is allowed for the value at the same grammatical position in the type II code clone. Consider the following code snippet:

```
if(a >= b) {
  c = d + 2;
  d = b - a;
}else
  c = d * b;
```

In this snippet, the `d = a - b;` in the third line of the earlier code snippet has been changed to `d = b - a;`. In general, the detection of type II code clones distinguishes which differently named identifiers are the same and which ones are different. Thus, the change in the order of the identifiers makes this code fragment not a type II code clone of the earlier code fragment. Similarly, in this snippet, the last line `c = +b;` of the earlier code snippet is changed such that the binary operator `+` is replaced with `*`. Because the detection of type II

code clones distinguishes between the binary operators +, -, *, and /, it is determined that the latter code snippet is not a type II code clone of the earlier code snippet.

However, in the process of detecting similar codes via the analysis of actual codes, there are cases in which it is necessary to ignore even these variations and track structural similarities. For example, suppose a piece of code is used in multiple places throughout the entire codebase. For various reasons, each piece of code may have acquired a slightly different form as the codebase was modified by multiple developers. If we find a problem with one piece of code, we may want to review all other similar types of code. Some developers may have accidentally changed the order of the variables or changed what should have been + to - during the editing process. If we attempt to detect type II code clones without ignoring these changes, we may overlook the real problem. In this case, by reducing the sensitivity of code clone detection to identifiers or operators, all binary operators are regarded as the same, or identifiers are regarded as the same as variable names, function names, etc. In this way, the changed code fragments can still be detected as type II code clones.

2.3. Type III Code Clone

Among the code clone types that consider text similarity, type III code clones are the type that allows the greatest variation. This type of code clone considers the overall similarity of code snippets and allows sentences to be changed, added, or deleted. Consider the following code snippet:

```
if (i >= j) {
  s = t + 1;
  t = i - j;
} else
  s = t + j;
```

The following code snippet is a type III code clone of the previous code snippet:

```
if (i >= j) {
  s = t + 1;
  u = v + t; //added.
  t = i - j;
} else
  s = t + j;
```

Although a new sentence has been added as the third line, the two code snippets show sufficient similarity, considering that the latter snippet has only six lines of code in total.

The detection of type III code clones is based on the similarity of the overall code, and thus, code fragments that, at first glance, appear to have many differences are also identified as code clones. Consider the following code snippets:

Code snippets 1 and 2 in Figure 1 show algorithms that traverse a tree depth-first and make a list of the labels of its nodes. If we simply compare the code snippets, the two look very different, except for some matching words. However, upon closer inspection, it can be seen that except for the addition of `sb.append(‘,’);` in code snippet 2, only the type or identifier was changed. Therefore, it is determined that of these two code snippets, one is a type III code clone of the other. If the third line of code snippet 2 had no added statements, then it could also be determined that of these two code snippets, one is a type II code clone of the other.

Code fragments are considered type III code clones if there is sufficient similarity between the actual code fragments, even if there are significant changes. Therefore, when the goal is specifically to detect this type of code clone, it is possible to find similar codes that exhibit differences beyond the various changes that occur during code development. The detection of type III code clones can also determine whether part of the code has been

divided into detailed pieces and copied. Therefore, many code clone detection studies have aimed at detecting type III code clones.

```
public void dfs(Node n, List<Node> nodes){
  nodes.add(n);
  for(Node c : n.cnodes()){
    dfs(c, nodes);
  }
}
```

(a) Code Snippet 1

```
public void dfs(Node n, StringBuffer sb){
  sb.append(n.getLabel());
  sb.append(",");
  for(Node c : n.cnodes()){
    dfs(c, sb);
  }
}
```

(b) Code Snippet 2

Figure 1. Example of a code snippet and its type III code clone.

2.4. Type IV Code Clone

Type IV code clones are discriminated based on semantic similarity, and not on the similarity of the code itself. Therefore, clones of this type may not have similar code structures and are less likely to be pieces of code copied from the original, which are generally agreed upon to be code clones. Type IV code clones can occur when similar logic is implemented in different forms for the same purpose. Consider this code that calculates a factorial:

```
int factorial(int n, int acc){
  if(n == 1)
    return acc;
  else
    return factorial(n - 1, n * acc);
}
```

The code snippet above forms a Type IV code clone with the following code snippet:

```
int fact(int n){
  int f = 1;
  for(int i = 2; i <= n; i++)
    f = f * i;
  return f;
}
```

The functions presented in the two code snippets have little lexical or syntactic similarities. Therefore, based on the criteria for type I–III code clones presented thus far, we can conclude that the two code snippets are not at all similar. However, regarding the meaning of the actual code, both code fragments perform the same function of calculating `n!` for a given `n`. Thus, the occurrence of the type IV code clone is determined based on the semantic similarity of this form.

Type IV code clones are very difficult to detect because they cannot be determined using simple textual, lexical, or grammatical similarities. The example presented here is intended to help in understanding type IV code clones by clearly indicating the purpose of the code. However, in general, research on detecting this type of code clone aims to find a piece of code that implements a similar type of logic, regardless of its purpose.

2.5. Gapped Clone

A gapped clone is formed when a certain difference occurs between two pieces of code; the code portion with this difference is called a gap [38]. When a piece of code is copied, three different types of gaps can occur:

- Rename and add code;
- Rename and delete code;
- Rename and change code.

Here, renaming includes cases in which there is no one-to-one correspondence with the original identifier, such as that shown in the last code snippet of Section 2.2. When codes are added or deleted, the sentences that have not been added or deleted will have the same grammatical structure. On the other hand, when a code is changed, the grammatical structure of the sentence in which the change occurred also changes. Gapped clones are included among type III code clones, which more clearly classify differences from the original code snippet. In addition, type III code clones consider the similarity of the whole code; therefore, even if a gapped clone has differences besides the type of gap presented, pieces of code with high similarity may belong to this group.

2.6. Reordered Clone

A reordered clone is formed when the order of parts of the code is changed. For example, consider a case in which the change in order does not change the data or control dependency of the code, such as that shown by two snippets of *bison* discovered by Komondoor and Horwitz [39].

If we swap the first and second lines of code of the two code snippets, and swap `fp1` and `fp2`, as shown in Figure 2, there is no difference, except that the remaining identifiers are different. Changing the order of the first two lines of code does not affect the execution of the rest of the code; therefore, semantically, the two code snippets have the same form of operation. Therefore, it can be considered that of these two code snippets, one is a type IV code clone of the other. Additionally, the difference between the two code snippets can be thought of as being due to identifier renaming and sentence change, and thus it can be determined that of the two codes, one is a type III code clone of the other.

```
++ fp1 = LA + i * tokensetsize;
++ fp2 = lookaheadset;
++ while (fp2 < fp3 )
++   *fp2++ |= *fp1++;
```

(a) Code Snippet 1

```
++ fp1 = base;
++ fp2 = F + j * tokensetsize;
++ while (fp1 < fp3)
++   *fp1++ |= *fp2++;
```

(b) Code Snippet 2

Figure 2. Example of a code snippet and its reordered clone.

3. Code Similarity Detection Techniques

In this section, we introduce code similarity detection techniques. First, the general code similarity detection process is examined overall, and various proposed methods for detecting similar codes are presented. These code similarity detection techniques can be classified in many ways; however, they are usually divided into hybrid methods that combine a variety of methods, such as text-based, token-based, tree-based, and PDG-based techniques, depending on how the code is expressed and processed. After examining these methods, we discuss the types of code clones that each method can detect and their respective limitations.

3.1. General Code Similarity Detection Process

Techniques for detecting similar codes are generally composed of pre-processing, transformation, and detection steps.

3.1.1. Pre-Processing

In the pre-processing step, the code is prepared for subsequent steps by processing into a desired shape. The code to be analyzed is selected from the entire codebase, and the unit to be analyzed is determined. For example, a code file containing constants to be used in a program may contain many similar codes; however, in many cases, these are not of

interest. Additionally, files that are not subject to analysis, such as those containing data other than the source code, are excluded. An important aspect of this step is determining the unit of code to be analyzed. The unit to be subjected to code similarity detection may be influenced by the purpose of detection and the type of code clone. To detect similar code by file, function, or sentence, the code is divided accordingly. These actions are often combined with the next step, which is transformation.

3.1.2. Transformation

In the transformation step, the pre-processed code is re-expressed as an expression method for code similarity detection. This is one of the key steps in code similarity detection, because the next step, i.e., detection, is significantly influenced by how the code is represented in the transformation step. Classification of code similarity detection techniques is also often based on this transformation step. Depending on the technique, the code is broken into strings of appropriate units, turned into a list of tokens through lexical analysis, and converted into an abstract syntax tree (AST) through parsing or into a PDG through more complex static/dynamic analysis. In this process, blank characters or comments that are not of interest in the analysis are removed.

In many cases, this transformation is processed mechanically using several proven programming language techniques; however, each technique makes some difference in the expression depending on the type of code clone to be detected. For example, in text-based techniques, each identifier is replaced and used for identifier normalization [10]. In addition, as discussed in Section 2, even when a code is expressed in token or tree form, the sensitivity of the expression is adjusted whenever the design of a technique is applied.

3.1.3. Detection

In the detection step, the transformed code representation is examined and searched for similar codes. This is the step of the process that demonstrates the largest differences between code similarity detection techniques; it can be said that each technique applies a distinct design for finding the target code clone. A matching algorithm is applied to the transformed code representation to create a list of clone pairs determined to be similar. The algorithms used for matching code expressions include hash value comparison [22,32,40], algorithms [10,16,33] that use suffix trees [41,42], and dynamic pattern matching (DPM). Recently, deep learning [34] and methods for extracting and comparing n-grams from lists of tokens [43] have also been used.

3.2. Text-Based Techniques

Text-based techniques consider code as a form of string enumeration. These techniques compare codes in units of predetermined strings to determine whether they match and generally operate in a way as to find the longest matching string. Text-based techniques tend to have the least transformation of code before the matching algorithm for detection is applied. However, because these techniques are purely string-based or use lexical analysis, the detected code clone may not correspond well with the grammatical structure of the code.

The advantage of text-based techniques is that they do not require complex code transformation; therefore, they are mostly language-independent and have fast execution times. In many cases, text-based techniques use the original code as a string without changing it into another form, and thus, the implementation of the technique itself is not affected by a change in language. Some techniques that use lexical analysis also use pattern matching to transform the necessary part of the code into a token form; consequently, the burden required to implement them in a new programming language is not significant.

A weakness of text-based techniques is that for similar code to be found, it must exhibit high textual similarity. Therefore, code clones detected using purely text-based techniques are mostly of type I. Nonetheless, when the detection targets discontinuous code clones via the division of the code into word- or line-based character strings, it becomes possible for type III code clones to be detected also, assuming that there is no identifier

change. However, these techniques, which require text matching in many cases, have limited detection performance and are unable to detect certain cases of similar codes that are affected by slight deformations.

One example of a purely text-based technique is Johnson's technique [11,12]. Its working principle is to divide the code line-by-line to extract and compare fingerprints. To extract the fingerprints, the Karp–Rabin algorithm [44,45] is used. To find additional matching codes, the conversion process removes spaces, etc., and replaces consecutive alphanumeric characters with a single character, `i`, for normalization. For example, if we have code `avg = sum/count`, it is converted to `i = i/i`. In this case, many false positives can occur, because the identifiers are mostly ignored. Nonetheless, including the condition that at least 50 lines of code must match, the number of falsely detected code clones can be reduced.

Baker's *Dup* [10,13] is a text-based technique that uses lexical analysis to apply regularization. This technique removes whitespace characters and comments; changes variables, functions, and type names to special parameters; and concatenates into a single line all the lines of the code unit that we want to subject to comparison. Subsequently, hash values are extracted from each line and compared, and a suffix tree algorithm is used to find the pair of longest matching lines. Although type II code clones can be detected using this method, this technique exhibits low overall detection performance and partial influence on the code style depending on the location of "`{`". To solve this problem, Baker devised a method that uses the token of each line when comparing the lines [46]. Because it uses tokens, it can be classified not only as a text-based technique but also as a token-based technique.

Another text-based technique is that of Ducasse et al. [14,15]. This technique reads a file, divides it into line units, removes spaces and comments, and applies DPM to detect similar code. The output is the line number of the clone pair, including a line deleted in the middle to indicate a gap clone. This method can be easily implemented in other programming languages because it does not perform language-dependent conversions such as parsing. However, this technique can only detect type I code clones. Furthermore, its language-independent characteristic makes it difficult to guarantee that the detected code clones truly represent meaningful similar codes.

3.3. Token-Based Techniques

Token-based techniques regard code as a sequence of tokens during the transformation and detection steps. After tokens are extracted from the code using lexical analysis, the way this information is used can vary significantly depending on the nature of the technique. Token-based techniques follow a method of finding matching sub-sequences in the entire token sequence and outputting the corresponding code parts as a code clone. Because codes are converted into token sequences, token-based techniques have an advantage in that they are not sensitive to minor changes, such as differences in code style, unlike text-based techniques, and have a high possibility of detecting meaningful similar codes because they include lexical analysis. However, because lexical analysis is required, token-based techniques cannot be applied completely independently of language, unlike text-based techniques, although it is still easier for token-based techniques to support new languages than it is for tree-based techniques.

One of the most famous token-based techniques is *CCFinder* [16], which supports a variety of programming languages, such as Java, C/C++, COBOL, VB, and C#. A new version, *CCFinderX* [17], was recently released [47]. CCFinder uses a lexical analyzer to extract tokens from each line of code and integrates tokens from the entire code into a sequence of tokens. Thereafter, the tokens are converted in two ways. The first is to modify the tokens according to the rules defined for each supported language. This is carried out to discriminate between codes with different grammatical structures but similar meanings. The second is a transformation that replaces identifiers corresponding to variables, functions, and type names with special tokens. The suffix tree-matching algorithm [48] is then used to find matches in this transformed token sequence.

Baker's techniques [10,13,46], which were introduced in Section 3.2, can also be classified as token-based techniques because they use lexical analysis. In particular, Baker applied a parameterized matching technique that replaces the same identifier with the same fully qualified name, as described in Section 2.2. For example, code snippets `max(a,b,a)` and `min(a,a,b)` on different lines can be converted into `method0(var0, var1, var0)` and `method0(var0, var0, var1)`, respectively. Using this transformation method, code clones can be detected by distinguishing between different parameter sequences, etc.

CP-Miner [18,19] uses a frequent subsequence mining algorithm called CloSpan [49] to find matching token sequences differently from the previously introduced techniques. A *closed subsequence* implies that the support of the subsequence is different from that of the sequence containing it, which CloSpan efficiently finds. CP-Miner uses extended CloSpan to detect gap clones without being affected by the addition or deletion of sentences. This overcomes the limitations of CCFinder and Baker's techniques, which are affected by the order in which tokens appear. CP-Miner has succeeded in detecting more than 150,000 clone pairs in a large-scale codebase, demonstrating that it can efficiently detect copy-pasted code, given a large amount of code.

Token-based techniques are also widely used in plagiarism detection because of their ease of supporting multiple languages and detecting code clones in multiple files. Among the techniques discussed in Section 4, those that employ token-based methods include *Winnowing* [50] (used by *MOSS*), *JPlag* [20], and *SIM* [21]. In the case of Winnowing, its method of using the fingerprint of the code in plagiarism detection is less affected by the partial addition, deletion, and modification of code; however, the fundamental limitations of token-based techniques still exist.

3.4. Tree-Based Techniques

Tree-based techniques convert target programs into parse trees or ASTs, identify common subtrees, and detect them as clones. For these, syntax analysis based on the grammar of the programming language in which the target program is written is required. Therefore, these techniques are dependent on the language used, and adding support for a new language requires more work than those needed for the text- or token-based techniques discussed earlier. However, if a parser for the language already exists, the process of applying the matching algorithm to the obtained AST will not change significantly. Adding support for a language that already has a parser is not much more difficult than for a token-based technique.

Unlike text-based or token-based techniques, which must undergo a process such as normalization to avoid being affected by specific identifiers, etc., AST reflects only the grammatical structure of the code (Figure A1), and thus this effect can be ignored without any special measures. However, if specific identification is required, such as for certain partial identifiers or types of operators, it is necessary to express these parts through the inclusion of additional information in the AST. A method widely used in actual implementations is to configure the label of the AST node with the addition of the specific property value shown in the actual code to the AST node type that represents the syntactic meaning, which is then used to find similar nodes in partial tree matching.

The greatest advantage of tree-based techniques is that the detected code clones are highly precise because two code fragments that are compared must have a similar grammatical structure to be considered a match. In token-based techniques, the comparison is between sequences of tokens, and thus, similar token sequences can be detected by chance. By contrast, in tree-based techniques, information for the grammatical structure is added; therefore, there is a high probability that the detected clone pairs have more grammatically significant similarities.

In addition, tree-based techniques have the advantage of capturing similarities between modified codes. Even if sentences are added, deleted, or changed, these changes are limited to a partial tree, and thus the remainder of the AST can be matched and determined to be a clone. For this reason, it is more resistant to transformations than token-based

techniques, where a significant number of tokens, and consequently the token sequence, can be changed when a sentence is changed. Therefore, tree-based techniques often aim to detect type II code clones without significant difficulty, and type III code clones.

However, a limitation of tree-based techniques is that they require parsing and tree-based comparisons to identify similar code. As previously mentioned, the need for parsing can be a stumbling block in supporting a new language, especially when a well-implemented parser does not yet exist. Moreover, the need for tree-based comparison implies that these techniques must employ algorithms applicable to two-dimensional data structures, which have a greater time complexity, instead of algorithms applicable to fast and efficient strings or sequences, which can affect their overall execution performance. This problem can affect the scalability of these techniques to large-scale codes. To overcome this, tree-based techniques also use a method for finding common parts that transform the AST into another form [23,24,32].

CloneDR [22] is a pioneer among tree-based techniques. After obtaining the AST from the target code, CloneDR analyzes the subtree. To overcome small transformations and detect similar subtrees, the subtrees are classified into buckets using a hash function. After a similarity metric is calculated for subtrees included in the same bucket, a similar subtree is detected if the similarity exceeds a certain threshold. This similarity metric is calculated using a simple method in which the proportion of nodes shared between two subtrees is divided by the proportion of all nodes, which is a method often used to calculate the similarity of trees. CloneDR has commercially available implementations [51] and supports the most popular languages, including C, C++, C#, Java, Python, and ECMAScript. Considering that the implementations of most techniques are not open, or that existing implementations support only some languages for proof of concept, CloneDR has an advantage in terms of accessibility, aside from its performance.

Another tree-based technique is that of Wahler et al. [23]. This technique further abstracts the AST, changes it into an XML expression [25], and uses a method for detecting similar codes based on frequent item set mining [26]. Another technique, proposed by Evans and Fraser [24], uses a higher level of abstraction. This technique parameterizes an arbitrary subtree to further abstract the AST.

For example, consider the following parameterized code clone:

```
if(s > ?)
  return t;
```

Two ways it can occur are as follows:

```
if(s > t)
  return t;

if(s > t * 2)
  return t;
```

In the first code snippet, the parameter `?` is simply replaced with `t`. Thus, it can be regarded as a leaf node change in the AST. However, in the second code snippet, `t * 2` corresponds to `InfixExpression`, which includes binary operators. In general, it appears in the form of a partial tree on the AST with child nodes corresponding to two operands attached to the parent node. In other words, because the parameter `?` in the aforementioned parameterized code clone is replaced with a partial tree, it can be said that it has a higher level of abstraction than if it were simply parameterized in the form of a one-to-one correspondence of nodes. Taking advantage of this, Evans and Fraser's technique can be used to detect gap clones.

3.5. Pdg-Based Techniques

PDG-based techniques analyze a given code to create a PDG representing the data flow and control flow of a program and then detect similar codes by applying an isomorphic

subgraph matching algorithm between their PDGs. Compared to the code representations of tree-based techniques, where the syntactic structure of the code remains intact, PDGs are more abstract and less subject to syntactic changes. Tree-based techniques can ignore small variations in detecting similar codes. However, as can be seen from the similarity metric used by CloneDR, described in Section 3.4, a certain level of syntactic similarity is still required to detect code clones. Beyond these limitations, PDG-based techniques can detect discontinuous clones or resequencing clones that lack grammatical similarity and are, thus, likely to detect type IV clones.

Although PDG-based techniques can overcome the weaknesses of other techniques and detect more complex clones, they have several limitations. The biggest problem is that regardless of the technique used, we need a tool that can write PDGs. Because the languages supported by currently available tools for creating PDGs are quite limited, this method of detecting code clones is not suitable when support for various languages is required. Additionally, because the overhead of creating a PDG is greater than that of other techniques that rely on tokens and trees, the overall performance of PDG-based techniques is inevitably inferior. Another problem is that because the PDG is a graph, detecting similar patterns is more difficult than when other representations are used. Compared to other code expressions for which efficient matching algorithms already exist, the isomorphic subgraph problem is an NP-complete problem, and although there are methods that can approximate it at a practical level, there is not yet a method to solve this problem efficiently. Therefore, it is difficult to apply this method to a large codebase, owing to its poor scalability.

Nevertheless, attempts have been made to develop PDG-based techniques that overcome the limitations of existing techniques. For example, Komondoor and Horowitz's *PDG-DUP* [27] detects code clones by finding PDG subgraphs using program slicing. Based on this, they also proposed a technique in which codes of similar functional units are automatically extracted by grouping the detected code clones while preserving the meaning of the original code [28,29]. On the other hand, Krinke proposed a method for detecting code clones that finds a maximal similar subgraph in a fine-grained PDG in an iterative manner [30]. Whereas techniques that consider only the grammatical structure of codes generally have a tradeoff, in that recall is lowered when precision is increased, the proposed technique has produced results with both high precision and recall when using a detailed PDG. *GPLAG* [31] is a plagiarism detection technique based on PDG. Its creators attempted to develop a technique that is resistant to attempts to avoid plagiarism detection using the characteristics of PDG-based techniques that are resistant to modification. In GPLAG, a search-space pruning method for plagiarism detection is used to compensate for the disadvantages of the PDG-based technique, which has poor scalability.

3.6. Hybrid Approaches

Hybrid methods refer to techniques that use a combination of the various code expression methods introduced thus far. However, these techniques may be classified as token- or tree-based techniques based on the main code expressions. Classification based on precise and rigorous code representation is not a simple problem, because many techniques proposed in practice attempt to combine various methods to employ the advantages of various code representations while avoiding their respective disadvantages.

Deckard [32] is a code clone detection technique, proposed by Jiang et al., that first writes an AST in a given code and then converts it into a node-type vector. The node-type vector contains elements for all the possible AST node types and the number of node types that actually appear. Figure 3 shows the AST written for the code snippet `if(a >= b) swap(a, b);` and its node-type vector. In the AST, nodes of type `if`, `infix`, and `func_call` appear only once each, whereas nodes of type `name` appear five times. These numbers appear as elements of a vector. On the other hand, the `return` type shown in blue is marked with the number 0 to indicate that it does not exist in the current AST, although it is a node type that can actually appear.

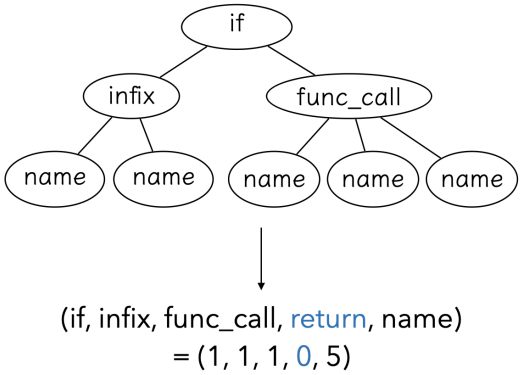


Figure 3. AST and node-type vector.

After Deckard converts the AST to an integer vector in this manner, it uses locality-sensitive hashing (LSH) [52] to place similar vectors into one bucket.

LSH is designed such that similar elements can be placed in the same bucket, unlike hash functions, which are generally designed to avoid collisions as much as possible. Figure 4 shows this characteristic of LSH in a comparison with normal hashing. Each circle indicates one element. The closer the distances of these elements to each other in the vector space, the greater their similarity. Whereas conventional hash methods place all elements in different buckets, LSH places two elements that are sufficiently close to each other in the first and last buckets. Using LSH in this manner, the effect of clustering using the Euclidean distance can be obtained very efficiently. Because ASTs placed in the same bucket are close in distance, they are detected as code clones.

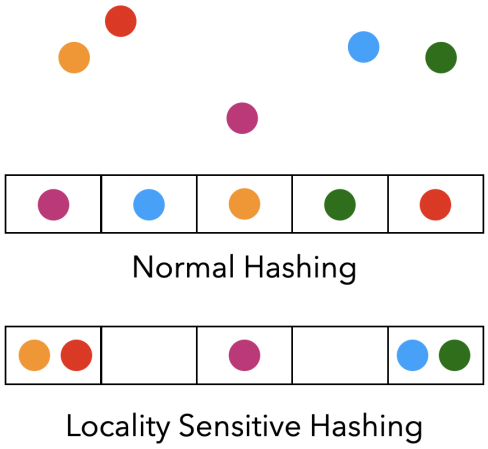


Figure 4. Difference between normal hashing and LSH.

The method of converting ASTs into node-type vectors, as employed by Deckard, and the method of using LSH for fast code similarity detection are useful in many ways. First, instead of comparing the ASTs, they are converted into integer vectors such that even if there are changes in some nodes, it will appear that some elements of the vector are changed. Therefore, if sufficiently close vectors are detected as code clones, similar codes can be found regardless of minor code changes. Furthermore, although this integer vector contains AST information, its structural information is abstracted into numbers. Even in the case of a reordered clone, in which the order of the code is changed, the range of similar codes that can be detected is wide because it has the same integer vector if its components are not changed. In addition, scalability is excellent because, using LSH, it is possible to quickly check whether there is a code similar to each vector instead of calculating the Euclidean distance of all pairs of vectors to detect nearby vectors.

White et al. [34] proposed a learning-based technique based on deep learning. This technique trains a model using information from the lexical analysis of a large amount of code and a model containing grammatical structures, by converting the AST into a vector. To identify code clones, this technique uses a method that converts a given code into an AST and infers whether there is similar code using the trained results. In their evaluation results, type I, II, III, and IV code clones were detected using a method that combined lexical and grammatical structure information. However, there are weaknesses in terms of efficiency; for example, the time required for learning was up to one hour, and the duration of the reasoning process for discriminating code clones ranged from several seconds to several tens of seconds. Furthermore, detection was performed only for clones in file and method units to secure code units for appropriate training.

Another hybrid method is a technique proposed by Koschke et al. [33], which visits AST nodes in a preordered sequence and arranges them in rows. For this, a suffix tree is constructed to derive the longest AST node sequence. As in tree-based methods, this technique constructs an AST but converts it into a token sequence and applies a suffix tree algorithm, like in token-based techniques. The grammatical structure information of the tree is used; however, instead of finding a matching partial tree to find similar code, this method converts it to linear data and processes it, which has the advantage of being able to detect similar codes very efficiently.

Tairass and Gray [35] proposed a technique for detecting functional-level code clones based on Microsoft's Phoenix framework. Like that by Koschke et al., this technique constructs a suffix tree using AST nodes and allows for analysis in linear time. However, this method requires a specific framework and is limited to detecting only exact or parameterized clones in functional units.

3.7. Evaluation of Code Similarity Detection Techniques

Thus far, we have examined a variety of code similarity detection techniques. In this section, the characteristics of each type of detection technique are summarized and evaluated, and trends in the research on code clone detection techniques are discussed.

3.7.1. Characteristics of Code Similarity Detection Techniques

Table 3 summarizes the characteristics of each code similarity detection technique. The level of abstraction impacts both the accuracy and generality of detection, whereas language dependency affects the diversity of detection targets. Efficiency governs the detection speed and resource utilization, whereas extensibility ensures applicability across various project scales and environments. These factors are pivotal in determining the quality and practicality of code similarity detection, rendering them indispensable in software development and maintenance. Given their significance, we have detailed each of these features in Table 3.

The level of abstraction indicates the abstraction of the code expression. The detected clone types are code clone types that are expected to be detected by each technique. Of course, because the types of code clones that can be detected differ even among techniques using the same code expression, this is a rough summary of the types that each technique can detect. Language dependency refers to the dependency of each technique on the language of the target code. The higher the language dependency, the greater the difficulty of adding support for a new language. Efficiency is generally a summary of the efficiency of techniques that use each code expression. Finally, extensibility refers to whether a technique of this type can be applied to a larger codebase or a larger set of code snippets.

Table 3. Characteristics of code similarity detection techniques.

Feature	Text-Based	Token-Based	Tree-Based	PDG-Based	Hybrid
Abstraction Level	low	middle	high	very high	middle
Detected Clone Types	Type I, II, parameterized clone, limited gap clone	Type I, II, parameterized clone, limited type III and gap clone	Type I, II, parameterized clone, limited type III and gap clone	Type I, II, III, IV and reordering clone	Type I, II, III, IV and parameterized, gapped, reordered clones, etc.
Language Dependency	low	middle	high	very high	high
Efficiency	high	high	middle	low	middle
Extensibility	high	high	middle	low	middle

Excluding hybrid techniques, the level of abstraction increases from left to right. However, language dependence tends to increase to obtain such code representation. Additionally, even if the level of abstraction is high, the amount of information contained in the actual code expression increases as the information is obtained through lexical, syntax, and static/dynamic analyses. Therefore, efficiency is partially reduced, and scalability is limited to properly handle this situation. Hybrid methods combine various techniques to properly compensate for and offset these strengths and weaknesses, thereby securing appropriate efficiency and scalability at an intermediate level of abstraction.

When deciding on which code similarity detection technique to use in practice, it is important to comprehensively evaluate the following aspects: (1) whether the type of code clone that can be detected by the technique to be used is suitable for the targeted similar code, (2) whether support for the target language is possible, and (3) whether the technique is appropriate for the size of the code to be inspected, i.e., is efficient and scalable.

3.7.2. Research Trends on Code Similarity Detection Techniques

Proposals for new code similarity detection techniques were actively made from the 1990s to the early 2000s. However, from the mid-2000s, proposals for new techniques began to decrease rapidly. Code clone detection tools such as CCFinder (2002) [16], CP-Miner (2006) [19], and Deckard (2007) [32], which are often used in research requiring code similarity detection, were all published in the early to mid-2000s. It is appropriate to consider that this trend resulted from a change in attitude toward code clones, rather than from technical limitations encountered in the development of code similarity detection techniques. In the past, after Fowler referred to code clones as *code smell* [53], there was a strong opinion that they impacted maintenance. Code clones are considered to occur because of reuse due to copying and pasting code; thus, if there is a defect in the code clone, the defect is propagated. Therefore, it is opined that there is a need to manage the same code during maintenance. However, as [54–57] revealed in research studies conducted to analyze the characteristics of code clones and their impact on maintenance, code clones tend to be more stable because they are reusable codes and are less likely to change to complete codes or require maintenance. In this way, it was discovered that developers have been managing them properly. Accordingly, interest in the development of a code clone detection technique, which is aimed at helping to detect and remove such code clones through refactoring or separate management, has decreased. In addition, already developed code clone detection techniques are diverse and have been evaluated to be of sufficiently usable performance. Therefore, unless a new requirement for detecting similar codes emerges, or an idea for a code similarity detection technique using a highly differentiated method is presented, it will be difficult for researchers to develop a code similarity detection technique with a high contribution.

4. Research and Tools Related to Plagiarism Detection

This section discusses various issues related to plagiarism detection and related research, and introduces techniques and tools that can be used to detect source-code plagiarism.

4.1. Definition of Plagiarism

Plagiarism and code clones are similar in that they occur in the form of similar codes and are generally known to be the result of copy–pasting; however, there are clear differences between the two. One of their most significant distinctions is that the proper reuse of code is encouraged in the software development process, whereas plagiarism is discouraged under all circumstances. As discussed in Section 3.7.2, the perception that code clones are unconditionally bad and should be eliminated is blurred. On the other hand, in the case of plagiarism, the plagiarist is likely already aware of the injustice of the act; therefore, efforts are made to modify the code copied in various forms to make it appear different from the original code.

However, to actually detect plagiarism, especially in an automated way, it is necessary to define plagiarism more clearly. In situations where code reuse is encouraged, particularly in programming and software development, simply referring to similar codes may not adequately address the problem of plagiarism. What we are interested in here is how plagiarism is recognized in studies related to actual plagiarism detection, rather than the dictionary definition of plagiarism. It is necessary to understand how plagiarism is defined in related studies to properly understand how they deal with it.

Many studies related to plagiarism [2,58–63] have defined plagiarism in various ways. What is commonly mentioned in these definitions is that plagiarism is an act of deceiving "**as if it were one's own**" without proper notation that the code was "**someone else's**". Interestingly, the definition of plagiarism itself does not explicitly mention similar codes. To determine whether someone else's code was imported, a method for implicitly checking for similar code is used.

To detect plagiarism according to this definition, it is first necessary to examine for similar codes and then check whether the similar codes have appropriate marks that indicate that the other codes have been reused. Therefore, the detection of similar code is only a prerequisite or side branch of the process, and the actual judgment and detection of plagiarism depend on whether code reuse is indicated. However, even if there is no notation, if several similar code snippets exist, the question of which one is the original remains. That is, because plagiarism is the use of someone else's work without the acknowledgment of reuse, plagiarism detection results in the problem of sorting out the original, which is important to identify, because the original, which is owned by its author, is considered to be the plagiarism-free version of the code.

A more unique definition is provided by Brixtel et al. [64], who stated that a document is considered suspicious if its similarity to other documents is significantly higher than the similarity between documents on average. Here, the term "document" refers to a document with source code; therefore, it is no different from a file containing the code. This definition directly refers to code similarity, and plagiarism-suspected code that conforms to this definition can be detected using the definition itself. Of course, detecting plagiarism-suspected code does not free the investigator from the problem of identifying the original.

In the end, the core of plagiarism detection, to properly detect plagiarism in any form, is to define a method for finding similar codes and identifying the original among them (if the original exists).

subsectionObfuscation Methods Plagiarism is an undesirable act, of which most actors are aware; thus, in many cases, obfuscation is applied to the copied code to avoid suspicion or detection of plagiarism. Obfuscation methods are intentionally used to avoid code similarity detection techniques and can range from a simple code style or grammatical structure change to complicated methods for implementing the same logic in a different form. Considering the types of code clones described in Section 2, we can estimate what

types of original and copied codes will appear when such obfuscation is applied, and what techniques should be used to detect them.

Table 4 is an extract of obfuscation methods reported in various plagiarism-related studies and organized by Novak et al. [6] into a table, presented with the corresponding code clone types. The "*" mark on a code clone type indicates that only some of the codes changed by the proposed obfuscation method are applicable or that an additional appropriate preprocessing/transformation process is required for detection.

Table 4. Obfuscation methods and corresponding code clone types.

Research	Obfuscation Method	Details	Clone Type
Faidhi and Robinson [65]	Code format change	Changing the code style, such as spaces	Type I
	Annotation change	Changing annotation within code	Type I
	Merge lines of code	Merging of multiple unit and type codes into one	Type I *
	Change to equivalent control structure	Changing the control type, such as changing `for` to `while`	Type III *, IV
Đurićand Gašević [66]	Translation of parts of the program	Translating variable names or comments from one language to another	Type I *, Type II
Donaldson et al. [67]	Change program output	Changing only the lines of code that indicate something	Type III *, IV *
	Identifier change	Changing identifiers such as variable names and function names to others	Type II
	Change order of independent lines	Changing the order of the lines without affecting the logic	Type III *, Reordering
	Disconnect the code	Splitting a unit of code, e.g., a line, function, class, etc., into multiple pieces.	Type I *, II *, III *
Prechelt et al. [20]	Change of constant value	Changing constant values that do not affect logic	Type II *
Grier et al. [68]	Add unnecessary lines	Adding a line that has no meaning to the execution	Type III
	Code simplification	Eliminating unnecessary or non-critical lines	Type III
Whale [69]	Replace function calls with code	Inserting code corresponding to the body of a function instead of calling it	Type IV
	Operators, accessors, and data type changes	Changing operators to equivalent or acceptable forms, modifiers, data types, e.g., `int` to `long`	Type II *, III *
	Combine copied code with own	Changing the syntax structure and logic by combining the copied code with own code	N/A

There are a total of 15 obfuscation methods outlined in Table 4. If a detection technique appropriate for type III code clones is used, similar codes can be expected to be detected, even in cases wherein 13 of the obfuscation methods, i.e., excluding two cases, are applied. Therefore, from a technical perspective, even if a certain level of obfuscation is applied, it is not very difficult to identify suspicious codes using code similarity detection techniques.

Cases that are difficult to find using general code similarity detection techniques include "replace function calls with code" and "combine copied code with original code".

In the case of replacing function calls with code, directly inserting the contents of a function into the code, instead of calling the function, may not be detected in code similarity detection, because the scale of code change with respect to the original can be very large, depending on the size of the function. Nonetheless, even in this case, plagiarism can be determined based on detecting the code of the copied and inserted function itself as similar code; specifically, a type I code clone. In the case of combining the copied code with one's code, the grammatical structure and logic are also changed; therefore, it is considered a very difficult case in which to determine plagiarism. This may or may not be detected by code similarity detection, depending on the extent to which the copied code maintains its original form. If the degree of similarity is remarkably lowered by transformation, it may not be possible to determine plagiarism, besides by understanding the actor's intention.

4.2. Source-Code Plagiarism Detection and Tools

As discussed in Section 4.1, plagiarism detection can be divided into two stages. In the first stage, suspected plagiarism candidates are identified based on searches for similar codes. In the second stage, it is necessary to determine whether a case of plagiarism from the plagiarism candidate group is actually plagiarism, that is, whether someone else's code is copied, and acknowledgment marks are omitted. A problem at the first stage is that plagiarists use various obfuscation methods to hide their intentions. However, this problem can be solved without significant difficulty using code similarity detection techniques, as discussed in Section 4.1. In the end, the problem that remains is determining whether a similar code is the result of an actual plagiarism act, which is difficult to solve mechanically using an automation-based technique.

Various source-code plagiarism detection techniques and tools focus on helping people make judgments in the second stage by solving the problem at the first stage, but not at the second stage. In other words, although a plagiarism detection technique or tool is named as such, there is essentially no significant difference between it and code similarity detection techniques. Of course, because the purpose is different, it may be possible to detect similar codes in cases wherein plagiarism detection techniques are less affected by known obfuscation methods.

To address this question, we considered the experimental results of the evaluation study conducted by Burd and Bailey on clone detection tools [70]. Among the experiments they conducted, we can find the precision and recall measurement results for plagiarism detection tools JPlag and Moss, which were briefly introduced in Section 3.3, together with those for clone detection tools CCFinder and CloneDR, which were discussed in Section 3. According to Novak et al., JPlag and Moss are the most frequently used plagiarism detection tools in plagiarism-related research studies [6]. In the original study, the results of *Cavet* [40], which are metric-based techniques, are also included. However, this study does not deal with metric-based techniques, and thus, these results are omitted. In the case of detecting similar codes, precision is determined by how many similar codes actually exist among the codes reported by the tools as similar codes, whereas recall is determined by how many of the total existing similar codes are detected by the tool. As mentioned in Section 3.5, a tradeoff exists between the two figures.

The results of gathering information on plagiarism detection tools and adding information on code clone detection tools are shown in Table 5. Supported languages denote representative languages supported by tools; OSS indicates open-source software. Rather than indicating whether the actual code was disclosed, the OSS column is marked with a `yes` if the detection technique itself was disclosed, and `no` if only a simple explanation was provided. Several plagiarism detection tools have not been included here but have been presented in several studies. However, except for the details, there are no significant differences from the tools presented in Table 5 because they are essentially tools for detecting similar codes.

Table 5. Information on plagiarism detection tools.

Tool	Support Language	OSS	Feature	URL
JPlag	C, C++, C#, Java, Scheme, Natural language	yes	Major language and natural language support	https://github.com/jplag/jplag [71]
MOSS	C, C++, Java, C#, Python, and many more	yes	Code submission required for service, separate license for commercial use	http://theory.stanford.edu/~aiken/moss [72]
CCFinderX	Java, C, C++, COBOL, VB, C#	yes	Provides an interactive interface	https://github.com/gpoo/ccfinderx [47]
CloneDR	Java, C#, C++, Python, JavaScript and many more	yes	Technique is public, but tools implemented are commercially available	http://www.semdesigns.com/products/clone/index.html [51]
Codequiry	Java, C, C++, Python, and many more	no	Commercial online services and provide comparison with code collected from the web	https://codequiry.com [73]
Copyleaks	Java, C, C++, Python, and many more	no	Commercial online services and provide improved results by machine learning	https://copyleaks.com/code-plagiarism-checker [74]

The first two tools in Table 5 are widely used plagiarism detection tools, the next two are code clone detection tools, and the latter two are commercial plagiarism detection tools available online. The strengths of JPlag and MOSS are that they were developed specifically for plagiarism detection from the beginning and that the actual techniques are transparently disclosed for non-commercial use, allowing the detection results to be predicted. In the case of CCFinderX and CloneDR, additional work may be required to use them for plagiarism detection, because they were developed for code clone detection rather than plagiarism detection. However, code similarity detection has the advantage of yielding better performance than those of the previous two tools. Codequiry and Copyleaks have advantages in that they provide a convenient interface, given that they are commercial products, and provide code similarity detection results using their own database. However, these tools are expensive and have a weakness in that the actual detection method or performance is not transparently disclosed.

As discussed in Section 4.1, the final judgment of plagiarism must be made by a person after seeing the code similarity detection result; against this human judgment, it is difficult to demonstrate overwhelmingly better results in plagiarism detection, regardless of which tool is selected. Therefore, it is desirable to select a tool that comprehensively considers other requirements, such as convenience and cost, rather than only plagiarism detection performance.

5. Discussion

The domain of code similarity and plagiarism detection faces challenging issues such as code obfuscation, semantic comprehension, cross-language detection, scalability, and false positives. Relevant questions that remain open within this domain pertain to the utilization of deep learning, integration of semantic analysis, management of code evolution, identification of behavioral cloning, assurance of privacy, establishment of benchmarks, and consideration of legal and ethical aspects. By addressing these challenges and questions, the precision and robustness of code similarity and plagiarism detection techniques can be enhanced, benefitting both research endeavors and practical applications in the field.

As discussed in Section 3, various techniques for detecting similar code demonstrate unique strengths and weaknesses, with each method excelling in identifying specific types of similar code instances. Therefore, the choice of an appropriate technique necessitates a comprehensive evaluation of the particular code type intended for identification. The progression of code similarity detection techniques has reached a mature stage, resulting in a reduced frequency of new methods emerging. Furthermore, the introduction of novel code similarity detection methods substantially enriches academic discourse only when they represent a revolutionary departure from existing methods.

In Section 4, we define plagiarism and thoroughly examine obfuscation strategies in the context of plagiarism detection. Additionally, we compiled and presented a range of tools and services designed for plagiarism detection. As indicated, plagiarism detection can be broadly categorized into two types: code similarity and comprehensive plagiarism detection. Most existing tools primarily concentrate on code similarity detection, which often requires human intervention for the final assessment of plagiarism. However, given the maturity of code similarity detection technology, a shift toward methods that offer more precise and user-friendly assessments of plagiarism based on the detected similar code is now needed to advance the field of plagiarism detection.

Research in the field of code clone and plagiarism detection has mostly stagnated since the mid-2010s, which can be attributed to several factors, including the maturation of existing technologies, the increasing complexity of the problem domain, challenges related to accessing and maintaining large-scale real-world code and plagiarism datasets, and a shift toward more specialized applications, such as security-focused plagiarism detection or code clone detection within specific programming languages or domains. However, it is important to clarify that this stagnation does not imply a complete cessation of research, but rather a concentration on refining mature technologies and exploring applications within specialized domains. The field continues to evolve, leaving room for future innovations, albeit at a more measured pace.

Several key areas remain open as important future research directions in the field of code clone and plagiarism detection. First, enhancing semantic comprehension in code detection and developing methods that better leverage semantic information. Second, implementing effective techniques to detect code similarities across multiple programming languages. Third, integrating code evolution and version control systems to incorporate code change history into detection methods. Fourth, detecting behavioral cloning, where code exhibits similar behavior but is not directly copied. Fifth, developing methodologies and tools to preserve privacy during the code detection process, especially in collaborative work environments. Sixth, establishing appropriate benchmarks, evaluation metrics, and standardized datasets to facilitate research comparisons and assessments. Lastly, addressing legal and ethical considerations concerning intellectual property and plagiarism, aiming for more transparent and equitable approaches to plagiarism detection while managing legal issues. These research directions are anticipated to contribute significantly to the advancement of improved methods and tools in the field of code clone and plagiarism detection, ultimately enhancing software quality and safeguarding intellectual property.

6. Conclusions

In this study, we consolidate the foundational theories and terminologies pertinent to code similarity detection techniques. Building on this groundwork, we introduce and compare various approaches for code similarity detection. Furthermore, our investigation extends the domain of plagiarism detection by exploring how code similarity detection techniques can be leveraged within this context. Nonetheless, we have not yet consolidated a summary of the findings from the studies we reviewed, presenting them cohesively to aid readers in making informed choices when selecting a detection technique, if necessary. Furthermore, we have not included individual practical examples from the studies we reviewed. These limitations will be subject to more comprehensive investigation and resolution in our future work.

The inherent challenge lies in the fact that the mechanized evaluation of plagiarism cannot rely merely on extracted technical information. A comprehensive assessment necessitates consideration of plagiarism policies, the context of code similarity generation, and the intent of the individual. This holistic approach underlines the prominence of plagiarism detection research within computer education conferences, as opposed to software and programming language forums. Therefore, directing research efforts toward the development of decision-making support systems that present diverse information for informed plagiarism evaluation is recommended over attempting to fully automate the process through purely technical means.

Author Contributions: G.L. and J.K. conceptualized the work, analyzed the data, and wrote the first draft. R.L. and R.-Y.J. performed the statistical analysis and contributed to the manuscript editing. M.-s.C. facilitated the acquisition of financial support for the project leading to this publication. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by a Research and Development project, Building a Data/AI-based Problem-solving System of Korea Institute of Science and Technology Information (KISTI), South Korea, under Grant K-23-L04-C05-S01.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Here, terms and concepts related to programming languages used to describe code similarity and plagiarism detection techniques are briefly described.

Appendix A.1. Token

The token indicates the type and attribute value of a lexeme obtained by lexical analysis of the code. Types of tokens include keywords or identifiers corresponding to reserved words in programming languages, operators, space characters, and literals. A lexical analyzer analyzes a given piece of code, or a stream of strings, and turns it into a sequence of tokens. For example, a token obtained by the lexical analysis of code, such as `return x * y + 2`, is `<return> <id, x>, <mul_op> <id, y> <add_op> <num, 2>`. In the case of the `id` token representing an identifier or the `num` token representing a number, an attribute value representing an actual lexical item is shown together. When checking code similarity, the array of tokens is compared using various methods.

Lexical analysis is performed by matching patterns defined by regular expressions; thus, defined tokens are obtained. Depending on the definition of the pattern, various parts of the code can be expressed in an appropriately abstracted form. For example, as discussed in Section 2.2, to treat all binary operators as equal, it is defined that both `<mul_op>` and `<add_op>` resolve to `<add_op>` in the example token. Thus, the code does not show any difference in the sequence of tokens, even if different lexical items appear, and this difference can be ignored.

Appendix A.2. Abstract Syntax Tree

An abstract syntax tree (AST) represents the syntactic structure of code in the form of a tree. AST is generally obtained by analyzing the structure of code based on the grammar of a programming language in the parsing process. In the case of a token, a pattern corresponding to a given code is searched, and a word corresponding to the vocabulary appearing at the corresponding position is created such that the strings of the actual code and the array of tokens are matched. However, AST expresses the grammatical structure of leaf nodes that correspond to actual codes through internal nodes. These internal nodes often do not appear as the corresponding strings in the code.

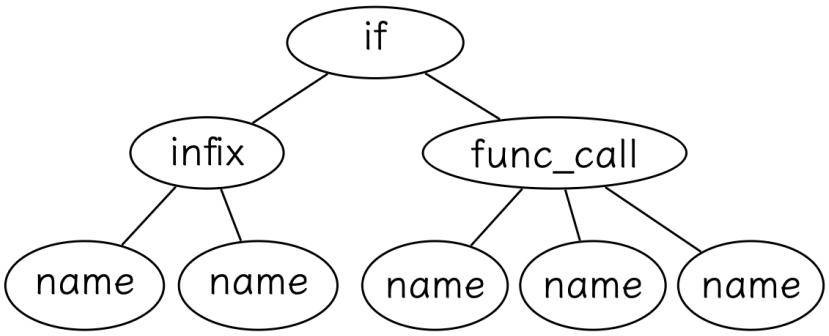


Figure A1. AST example.

Figure A1 is the AST obtained by parsing the code snippet `if(a >= b) swap(a, b);` The five `name` nodes corresponding to the leaf nodes correspond to the variable and function names of `a, b, swap, a, b`. Unlike the leaf node, where there is a clear correspondence, the inner node `func_call` does not have a clear code part corresponding only to this node. However, this node shows information that its child nodes, `swap, a, b`, form code corresponding to a function call.

Compared to the list of tokens, the AST even shows the grammatical structure of the code, so we can compare codes using more information. Like tokens, in the case of AST, it is possible to create a code expression whose sensitivity varies depending on whether to use only the type of node, `if, infix, name`, to distinguish each node, or whether to consider the identifier represented by this node in a node such as `name`.

Appendix A.3. Program Dependence Graph

A program dependence graph (PDG) is a graph representing the data and control dependencies of codes that displays the semantic relationships between code parts. Starting from the entry point of the code, we draw the edges from one code part A to another code part B based on the code using dependencies.

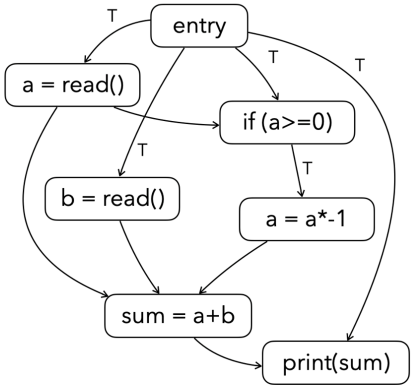
Figure A2b presents the PDG obtained by analyzing the code in Figure A2a. Each node corresponds to one sentence except for the `entry` node, which indicates the starting point of the code. Among the edges, those marked with `T` represent control dependencies, meaning that if the condition is true, the program control moves through the corresponding edge. For example, the first three lines of the code are always executed after execution starts, unless an exception occurs; therefore, the `entry` node, which indicates code entry, and the trunk line, which indicates execution if true, are connected. Similarly, the node indicating `if(a >= 0)` in the third line and the node corresponding to the fourth line, which is executed only when `a >= 0` is true, are also connected. In the node representing `sum = a + b` in line 5, the value of `sum` in this node depends on the values of `a, b`; therefore, the values of these two variables can change. These three nodes are connected by an edge, which indicates a data dependency.

As the PDG expresses semantic relationships regardless of the grammatical structure of the code, changes in the structure of the code do not affect the PDG if the meaning is maintained. For example, PDG appears the same as in Figure A2b, even if `read()` in the first two lines of Figure A2a code is replaced with a different variable or integer literal, or `sum = a + b` and `print(sum)` in lines 5–6 are changed to `avg = (a + b)/2` and `print(avg)`.

Because analyses such as program slicing are required to obtain a PDG from the code, it is more expensive than obtaining an abstract syntax tree based on lexical analysis or grammar using simple pattern matching. In addition, compared to displaying code as a list of tokens or expressing a grammatical structure in the form of a tree, displaying it as a graph is more difficult and has higher costs.

```
a = read();
b = read();
if (a >= 0)
  a = a * -1;
sum = a + b;
print(sum);
```

(a) Code example



(b) PDG example

Figure A2. PDG example.

Appendix A.4. Level of Abstraction

The level of abstraction, whether high or low, denotes the extent to which a specific task or concept is articulated in broader or high-level terms, distancing itself from specific details. The assessment of the level of abstraction can be based on the following criteria:

1. Concreteness vs. generality: Low-level abstraction is concrete and emphasizes specific details, whereas high-level abstraction is more general and pertains to overarching principles, patterns, or concepts. For instance, working with specific regular expression patterns exemplifies low-level abstraction, while addressing general patterns like email addresses represents high-level abstraction.
2. Specific details vs. core concepts: low-level abstraction refers to specific implementation details of a task, whereas high-level abstraction addresses core concepts and principles, offering ideas or methodologies that are applicable in diverse situations.
3. Complexity vs. simplicity: Low-level abstraction is employed when confronting intricate and detailed tasks. Conversely, high-level abstraction offers simpler and more abstract methods to conceal complexity.
4. Dependency on specific tools or technologies vs. independence: low-level abstraction may rely heavily on specific tools or technologies, whereas high-level abstraction is more autonomous and can be employed across various tools or environments.
5. Detailed implementation vs. general interface: low-level abstraction often encompasses specific implementation details, whereas high-level abstraction is typically presented as an interface or abstract concept, concealing specific implementation factors.

The selection of an appropriate level of abstraction hinges upon the complexity of the task, objectives, context, and requirements. Certain tasks demand low-level abstraction, whereas others may favor high-level abstraction. Consequently, the level of abstraction can vary depending on the specific use.

References

1. Nickolay , V.; Petr, G.; Andrey, F. A Machine Learning-Based Approach for Detecting Plagiarism in Source Code. In Proceedings of the 2020 ACAI '20: 3rd International Conference on Algorithms, Sanya, China, 24–26 December 2020.
2. Georgina, C.; Mike, J. An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis. *IEEE Trans. Comput.* **2011**, *61*, 379–394.
3. Anala, A.P.; Gaurav, T. Review of Plagiarism Detection Technique in Source Code. In Proceedings of the International Conference on Intelligent Computing and Smart Communication 2019, Singapore, 20 December 2019.
4. Oscar, K.; Simon; William, C. Similarity Detection Techniques for Academic Source Code Plagiarism and Collusion: A Review. In Proceedings of the 2019 IEEE International Conference on Engineering, Technology and Education (TALE), Yogyakarta, Indonesia, 10–13 December 2019.
5. Hayden, C.; Yuqing, L.; Shamus, P.S. Academic Source Code Plagiarism Detection by Measuring Program Behavioral Similarity. *IEEE Access* **2021**, *9*, 50391–50412.

6. Matija, N.; Mike, J.; Dragutin, K. Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review. *ACM Trans. Comput. Educ.* **2019**, *19*, 27.
7. Simon; Karnalim, O.; Sheard, J.; Dema, I.; Karkare, A.; Leinonen, J.; Liut, M.; McCauley, R. Choosing Code Segments to Exclude from Code Similarity Detection. In Proceedings of the ITiCSE-WGR'20: Working Group Reports on Innovation and Technology in Computer Science Education, Trondheim, Norway, 17–18 June 2020.
8. Oscar, K.; Setia, B.; Hapnes, T.; Mike, J. Source Code Plagiarism Detection in Academia with Information Retrieval: Dataset and the Observation. *Inform. Educ.* **2019**, *18*, 321–344.
9. Arkan, K.S.S.; Abdul, K.G. Plagiarism detection in learning management system. In Proceedings of the 2017 8th International Conference on Information Technology (ICIT), Amman, Jordan, 17–18 May 2017.
10. Baker, B.S. A program for identifying duplicated code. *Comput. Sci. Stat.* **1993**, 1–9.
11. Johnson, J.H. Identifying redundancy in source code using fingerprints. In Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering, Toronto, ON, Canada, 24–28 October 1993; Volume 1, pp. 171–183.
12. Johnson, J.H. Substring Matching for Clone Detection and Change Tracking. *ICSM* **1994**, 120–126.
13. Baker, B.S. Parameterized diff. In Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, USA, 17–19 January 1999.
14. Ducasse, S.; Rieger, M.; Demeyer, S. A language independent approach for detecting duplicated code. In Proceedings of the IEEE International Conference on Software Maintenance—1999 (ICSM'99), 'Software Maintenance for Business Change' (Cat. No. 99CB36360), Oxford, UK, 30 August–3 September 1999.
15. Ducasse, S.; Nierstrasz, O.; Rieger, M. *Lightweight Detection of Duplicated Code. A Language-Independent Approach*; Institute for Applied Mathematics and Computer Science, University of Berne: Bern, Switzerland, 2004.
16. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* **2002**, *28*, 654–670. [CrossRef]
17. Kamiya, T. CCFinderX: An Interactive Code Clone Analysis Environment. In *Code Clone Analysis*; Katsuro, I., Chanchal, K.R., Eds.; Springer: Singapore, 2021; pp. 31–44.
18. Li, Z.; Lu, S.; Myagmar, S.; Zhou, Y. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. *OSdi* **2004**, *4*, 289–302.
19. Li, Z.; Lu, S.; Myagmar, S.; Zhou, Y. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng.* **2006**, *32*, 176–192. [CrossRef]
20. Prechelt, L.; Malpohl, G.; Philippsen, M. Finding plagiarisms among a set of programs with JPlag. *J. UCS* **2002**, *8*, 1016–1038.
21. Gitchell, D.; Tran, N. Sim: A utility for detecting similarity in computer programs. *ACM Sigcse Bull.* **1999**, *31*, 266–270. [CrossRef]
22. Baxter, I.D.; Yahin, A.; Moura, L.; Sant'Anna, M.; Bier, L. Clone detection using abstract syntax trees. In Proceedings of the International Conference on Software Maintenance, Bethesda, MD, USA, 20 November 1998.
23. Wahler, V.; Seipel, D.; Wolff, J.; Fischer, G. Clone detection in source code by frequent itemset techniques. In Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation, Chicago, IL, USA, 16 September 2004.
24. Evans, W.S.; Fraser, C.W.; Ma, F. Clone detection via structural abstraction. *Softw. Qual. J.* **2009**, *17*, 309–330. [CrossRef]
25. Fischer, G. Simplifying source code analysis by an XML representation. *Softwaretech. Trends* **2003**, *23*, 49–50.
26. Han, J.; Pei, J.; Kamber, M. *Data Mining: Concepts and Techniques*, 3rd ed.; Elsevier: Cambridge, MA, USA, 2011.
27. Komondoor, R.; Horwitz, S. Using slicing to identify duplication in source code. In Proceedings of the International Static Analysis Symposium, Paris, France, 16–18 July 2001.
28. Komondoor, R.; Horwitz, S. Semantics-preserving procedure extraction. In Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, MA, USA, 19–20 January 2000.
29. Komondoor, R.; Horwitz, S. Effective, automatic procedure extraction. In Proceedings of the 11th IEEE International Workshop on Program Comprehension, Portland, OR, USA, 10–11 May 2003.
30. Krinke, J. Identifying similar code with program dependence graphs. In Proceedings of the Eighth Working Conference on Reverse Engineering, Stuttgart, Germany, 2–5 October 2001.
31. Liu, C.; Chen, C.; Han, J.; Yu, P.S. 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, Philadelphia, PA, USA, 20–23 August 2006.
32. Jiang, L.; Misherghi, G.; Su, Z.; Glondu, S. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 20–26 May 2007.
33. Koschke, R.; Falke, R.; Frenzel, P. Clone Detection Using Abstract Syntax Suffix Trees. In Proceedings of the 2006 13th Working Conference on Reverse Engineering, Pittsburgh, PA, USA, 7–11 November 2005.
34. White, M.; Tufano, M.; Vendome, C.; Poshyvanyk, D. Deep learning code fragments for code clone detection. In Proceedings of the 1st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 3–7 September 2016.
35. Tairas, R.; Gray, J. Phoenix-based clone detection using suffix trees. In Proceedings of the 44th Annual Southeast Regional Conference, Melbourne, FL, USA, 10–12 March 2006.
36. Davey, N.; Barson, P.; Field, S.; Frank, R.; Tansley, D. The development of a software clone detector. *Int. J. Appl. Softw. Technol.* **1995**, *1*, 219–236.

37. Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.* **2007**, *33*, 577–591. [CrossRef]
38. Yasushi, U.; Toshihiro, K.; Shinji, K.; Katsuro, I. On detection of gapped code clones using gap locations. In Proceedings of the Ninth Asia-Pacific Software Engineering Conference, Gold Coast, QLD, Australia, 4–6 December 2002.
39. Komondoor, R.; Horwitz, S. Tool demonstration: Finding duplicated code using program dependencies. In Proceedings of the European Symposium on Programming, Genoa, Italy, 2–6 April 2001; pp. 383–386.
40. Mayrand, J.; Leblanc, C.; Merlo, E. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In Proceedings of the 1996 International Conference on Software Maintenance, Monterey, CA, USA, 4–8 November 1996.
41. McCreight, E.M. A space-economical suffix tree construction algorithm. *J. ACM (JACM)* **1976**, *23*, 262–272. [CrossRef]
42. Kosaraju, S.R. Faster algorithms for the construction of parameterized suffix trees. In Proceedings of the IEEE 36th Annual Foundations of Computer Science, Milwaukee, WI, USA, 23–25 October 1995; pp. 631–638.
43. Nakagawa, T.; Higo, Y.; Kusumoto, S. NIL: Large-Scale Detection of Large-Variance Clones. In Proceedings of the ESEC/FSE 2021: 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021.
44. Karp, R.M. Combinatorics, complexity, and randomness. *Commun. ACM* **1986**, *29*, 98–109. [CrossRef]
45. Karp, R.M.; Rabin, M.O. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* **1987**, *31*, 249–260. [CrossRef]
46. Baker, B.S. On finding duplication and near-duplication in large software systems. In Proceedings of the 2nd Working Conference on Reverse Engineering, Toronto, ON, Canada, 14–16 July 1995.
47. CCFinderX. Available online: https://github.com/gpoo/ccfinderx (accessed on 11 August 2023).
48. Gusfield, D. Algorithms on stings, trees, and sequences: Computer science and computational biology. *ACM Sigact News* **1997**, *28*, 89–180. [CrossRef]
49. Yan, X.; Han, J.; Afshar, R. Clospan: Mining: Closed sequential patterns in large datasets. In Proceedings of the 2003 SIAM International Conference on Data Mining, San Francisco, CA, USA, 1–3 May 2003.
50. Schleimer, S.; Wilkerson, D.S.; Aiken, A. Winnowing: Local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA, 9–12 June 2003.
51. CloneDR. Available online: http://www.semdesigns.com/products/clone/index.html (accessed on 11 August 2023).
52. Datar, M.; Immorlica, N.; Indyk, P.; Mirrokni, V.S. Locality-sensitive hashing scheme based on p-stable distributions. In Proceedings of the Twentieth Annual Symposium on Computational Geometry, New York, NY, USA, 8–11 June 2004.
53. Fowler, M.; Beck, K. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley Professional: Boston, MA, USA, 1999.
54. Aversano, L.; Cerulo, L.; Di Penta, M. How Clones are Maintained: An Empirical Study. In Proceedings of the CSMR'07: 11th European Conference on Software Maintenance and Reengineering, Amsterdam, The Netherlands, 21–23 March 2007.
55. Duala-Ekoko, E.; Robillard, M.P. Tracking Code Clones in Evolving Software. In Proceedings of the ICSE 2007 29th International Conference on Software Engineering, Minneapolis, MN, USA, 20–26 May 2007.
56. Kapser, C.J.; Godfrey, M.W. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empir. Softw. Eng.* **2008**, *13*, 645–692. [CrossRef]
57. Rahman, F.; Bird, C.; Devanbu, P. Clones: What is that smell? In Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), Cape Town, South Africa, 2–3 May 2010.
58. Kustanto, C.; Liem, I. Automatic source code plagiarism detection. In Proceedings of the 2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, Daegu, Republic of Korea, 27–29 May 2009.
59. Hage, J.; Vermeer, B.; Verburg, G. Plagiarism Detection for Haskell with Holmes. In Proceedings of the CSERC '13: The 3rd Computer Science Education Research Conference, Arnhem, The Netherlands, 4–5 April 2013.
60. Kaučič, B.; Sraka, D.; Ramšak, M.; Krašna, M. Observations on plagiarism in programming courses. In Proceedings of the 2nd International Conference on Computer Supported Education, Valencia, Spain, 7–10 April 2010.
61. Konecki, M.; Orehovacki, T.; Lovrencic, A. Detecting computer code plagiarism in higher education. In Proceedings of the ITI 2009 31st International Conference on Information Technology Interfaces, Cavtat, Croatia, 22–25 June 2009.
62. Mariani, L.; Micucci, D. AuDeNTES: Automatic detection of tentative plagiarism according to a reference solution. *ACM Trans. Comput. Educ. (TOCE)* **2012**, *12*, 1–26. [CrossRef]
63. Hage, J.; Rademaker, P.; van Vugt, N. Plagiarism detection for Java: A tool comparison. In Proceedings of the Computer Science Education Research Conference, Heerlen, The Netherlands, 7–8 April 2011.
64. Brixtel, R.; Fontaine, M.; Lesner, B.; Bazin, C.; Robbes, R. Language-independent clone detection applied to plagiarism detection. In Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, Timisoara, Romania, 12–13 September 2010.
65. Faidhi, J.A.W.; Robinson, S.K. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Comput. Educ.* **1987**, *11*, 11–19. [CrossRef]
66. Đurić, Z.; Gašević, D. A source code similarity system for plagiarism detection. *Comput. J.* **2013**, *56*, 70–86. [CrossRef]
67. Donaldson, J.L.; Lancaster, A.; Sposato, P.H. A plagiarism detection system. In Proceedings of the Twelfth SIGCSE Technical Symposium on Computer Science Education, St. Louis, MO, USA, 26–27 February 1981.
68. Grier, S. A tool that detects plagiarism in Pascal programs. *ACM Sigcse Bull.* **1981**, *13*, 15–20. [CrossRef]

69. Whale, G. Identification of program similarity in large populations. *Comput. J.* **1990**, *33*, 140–146. [CrossRef]
70. Burd, E.; Bailey, J. Evaluating clone detection tools for use during preventative maintenance. In Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation, St. Montreal, QC, Canada, 1 October 2002.
71. JPlag. Available online: https://github.com/jplag/jplag (accessed on 11 August 2023).
72. MOSS. Available online: http://theory.stanford.edu/~aiken/moss (accessed on 11 August 2023).
73. Codequiry. Available online: https://codequiry.com (accessed on 11 August 2023).
74. Copyleaks. Available online: https://copyleaks.com/code-plagiarism-checker (accessed on 11 August 2023).