IJEE
International Journal of
Engineering Education (IJEE)
e-ISSN : 2540-9808

Novel Code Plagiarism Detection Based on Abstract Syntax Tree and Fuzzy Petri Nets

Yu-Ying Wang[1], Rong-Kuan Shen[2], Gwo-Jen Chiou[3], Cheng-Ying Yang[4], Victor R.L. Shen[5], Farica Perdana Putri[5]

[1]Department of Applied Foreign Languages, Jinwen University of Science and Technology, Taiwan
[2]Department of Japanese Language and Literature, Shih Hsin University, Taiwan
[3]Department of Electrical Engineering, National Formosa University, Taiwan
[4]Department of Computer Science, University of Taipei, Taiwan
[5]Department of Computer Science and Information Engineering, National Taipei University, Taiwan

Email: rlshen@mail.ntpu.edu.tw

ABSTRACT - *Those students who major in computer science and/or engineering are required to design program codes in a variety of programming languages. However, many students submit their source codes they get from the Internet or friends with no or few modifications. Detecting the code plagiarisms done bystudents is very time-consuming and leadsto the problems of unfair learning performance evaluation. This paper proposes a novel method to detect the source code plagiarisms by using a high-level fuzzy Petri net (HLFPN) based on abstract syntax tree (AST). First, the AST of each source code is generated after the lexical and syntactic analyses have been done. Second, token sequence is generated based on the AST. Using the AST can effectively detect the code plagiarism by changing the identifier or program statement order.Finally, the generated token sequences are compared with one another using an HLFPN to determine the code plagiarism. Furthermore, the experimental results have indicated that we can make better determination to detect the code plagiarism.*

Keywords: *Computer Science Education, Source Code Plagiarism, Lexical Analysis, Syntactic Analysis, Abstract Syntax Tree, Petri Net.*

Submission: May 29, 2018 Correction: July 29, 2018 Accepted: Februari 27, 2019

Doi: http://dx.doi.org/10.12777/ijee.1.1.46-56

[How to cite this article: Wang, Y. Y., Shen, R. K., Chiou, G. J., Yang, C. Y., Shen, V. R. L., Putri, F. P. (2019), Novel Code Plagiarism Detection Based on Abstract Syntax Tree and Fuzzy Petri Nets. *International Journal of Engineering Education,* 1(1), 46-56. doi: http://dx.doi.org/10.12777/ijee.1.1.46-56]

I. INTRODUCTION

Developing a program is required to be done by those students who major in computer science and/or engineering in colleges. They are required to design program codes in a variety of programming languages. Therefore, the teachers provide plenty of coding assignments. Some of them could be done in the classroom or at home. Such coding assignments are very helpful to learn programming languages and to acquire programming skills. With the rapid development of open source code, many documents and source codes are available on the Internet and easy to access [1]. Thus, when the coding assignments are collected and examined, many of these tasks could have the same or similar source codes.

Students submit the source code they get from the Internet or friends with no or little modification. Plagiarism has been defined as someone handing in a report or document as his/her own original work which was, in fact, written or created by someone else [2]. Detecting code plagiarisms done by the students is very time-consuming for the teachers, who actually need to take their time to prepare other teaching materials or assignments. Therefore, for the purpose of faster finding code plagiarisms, researchers have proposed some methods or tools to detect the code plagiarisms.

Furthermore, the source code plagiarism often deals with the uncertainty in code similarity. It is because each student has designed his/her own algorithm with different complexities, performances, etc., but he/she solves the same problem. It should handle the vagueness in which students express their solutions. This paper proposes a novel method of detecting the source code plagiarism by using a high-level fuzzy Petri net (HLFPN) based on abstract syntax tree (AST), which is not influenced by changing the identifier or program statement order.

The HLFPN was adopted to deal with the uncertainty or vagueness in code plagiarism detection. First, the abstract syntax tree of each source code is generated after the lexical and syntax analyses have been done. Second, the token sequence is generated based on an AST. Finally, the generated token sequences are compared with one another using a high-level fuzzy Petri net to determine the code plagiarism.

In Section 2, we provide a literature review on the plagiarism in computer science, source code plagiarism detection techniques, abstract syntax tree, sequence alignment, and a high-level fuzzy Petri net. In Section 3, the framework of the proposed AST and HLFPN-based code plagiarism detection system is described. The experimental results and analyses are presented in Section 4. Finally, the conclusion and future work are summarized in Section 5.

II. LITERATURE REVIEW

In this section, we first describe the plagiarism in computer science, the development of source code plagiarism detection techniques, the abstract syntax tree, and the sequence alignment. Then, we describe some basic definitions regarding a high-level fuzzy Petri net and a fuzzy reasoning algorithm to determine the decision output.

Plagiarism in Computer Science

The problem of students plagiarizing is ongoing in educational institutions and is not confined to the submission of essays and other text-based assignments. It is also an issue within the computing disciplines, where students must write the program code that is assessed for correctness and quality [8].

Jones [9] described a definition of plagiarism detection in computer science, characterizing it as a problem of pattern analysis, based on plagiarizing transformations which have been applied to a source file. Such transformations are presented as follows [9]:

1. verbatim copying,
2. changing comments,
3. changing white space and formatting,
4. renaming identifier,
5. reordering code blocks,
6. reordering statements within code blocks,
6. changing the order of operands/operators in expressions,
7. changing data types,
8. adding redundant statements or variables,
9. replacing control structures with equivalent structures.

Kikuchi, et. al. [2] defined the source code plagiarism as someone handing in a report or documentation as his/her own original work which was, in fact, written or created by someone else. Students copy other work and then make no or few modifications.

Source Code Plagiarism Detection Approaches

According to Zhao, et.al. [10], in the early times of the concepts of plagiarism, the maindetecting technology of a copy was based on the file. This method is very simple and determines whether the two files are similar by comparing the calculated values of the files [10]. But, it can just detect the code plagiarism without transformation or modification. Roy and Cordy [11] classified six different detection techniques as follows:

Text-based Approach

This approach treats the source code as a pure text. The target source program is considered as a sequence of lines or strings. Two code fragments are compared with each other to find the sequences of same text/strings. If all strings are the same, we consider the two source codes which are homologous.

There are several problems that can arise in a line-by-line detection technique, presented as follows:

1. Line break: code portions with line break relocation are not detected as clones or detected as shorter clones.

2. Identifier changes: changes of identifier names may not be handled in line-by-line technique as it compares the similarity of texts/strings.

3. Parenthesis removal or addition for a single statement: For instance, if…else… statement or for statements can be written with or without begin-end brackets (“{“ and “}”). In the line-by-line technique, the presence of “{“ and “}” pair in one code segment but not in the other one may be detected as a distinct fragment. Therefore, it is obvious that different kinds of coding style can create problems in the line-by-line technique.

4. Transformations: any source code transformation is not suitable in the line-based approach.

Token-based Approach

In the token-based approach, the entire source code is lexed/parsed/transformed into a sequence of tokens. Each word in the source code is treated as a token. This sequence is then examined to find the duplicated subsequences of tokens. This technique is more robust against code changes such as formatting and spacing compared to the text-based approach.

The leading tool of this approach is CCFinder [14]. Kamiya used the lexical rules to convert each word in the source code into a token which can eliminate the impact of changes of variable and function names. He also removed the white spaces between tokens during lexical analysis, and used it later to reconstruct the original source code. Other detecting tools, such as CP-Miner, JPlag, and Winnowing are all token-based ones [15]-[17]. However, all the tools cannot detect the modification of renaming, reordering, and inserting null strings [10].

Tree-based Approach

The tree-based approach uses an abstract syntax tree (AST) created by parsing the source code, and then compares with one another by using the tree structure. The AST-based approach disregards the information about identifiers (in order to make codes differing on variable names which appear the same on ASTs), and ignoresthe data flows.So,it becomes fragile to statement reordering [11]. This approach is more durable against any modification made by a plagiarist than the previous approaches. Therefore, some algorithms have been proposed in [2],[10],[18-21].

Feng, et. al. [20] proposed an algorithm to detect the code plagiarism based on the AST. The algorithm can detect the plagiaristic cases by comparing the hash value of the node to detect the plagiarism between two source code files [20]. Their algorithm can effectively detect the following plagiarism cases: changing the variable name, reordering a sequence of expression evaluation, changing some parts of the code statements, and so on.

PDG-based Approach

The Program Dependency Graph (PDG)-based approach contains the control flow and data flow information of a program and hence carries semantic information. Once a set of PDGs are obtained from a subject program, the isomorphic subgraph matching algorithm is applied for finding the similar subgraphs which are returned as clones [11].

As Bellon, et. al. [22] stated, the advantage of PDG-based detection is that it can detect non-contiguous code clones, whereas other detection techniques are less effective in detecting them. A non-contiguous code clone is the one having elements that are not consecutively located on the source code [23]. It has been reported that, after copying and pasting a code fragment, the pasted code is sometimes incorrectly changed or forgotten to be changed [24]. On the other hand, the PDG-based code clone detection also has some disadvantages. For example, the ability to detect contiguous code clones is inferior to other techniques,and the application of PDG-based detection to practical software systems is not feasible because to do so is time consuming [25]-[26].

Metrics-based Approach

These approaches gather different metrics for code fragments and compare these metrics vectors instead of comparing code fragments directly. There are several detection techniques that use various software metrics for detecting similar code fragments. First, a set of software metrics called fingerprinting functions are calculated for one or more syntactic units such as a class, a function, a method, or even a statement.Then the metric values are compared with one another to find the clones over these syntactic units.

Mayrand, et. al. [27] proposed the code clone identification based on metrics extracted from the source code using the tool, DatirxTM. This technique uses 21 function metrics grouped into four points of comparison: name, layout, expressions, and control flow. After the metrics are obtained, they defined eight strategies in identifying clones. They have found that the technique is useful in improving the maintainability of a software system by managing and removing the source code function clones. However, the computational cost is polynomial and the main cost in conducting the experiments is the software measurement. Thus, they have to optimize and enhance the technique in order to apply it to a very large scale system.

Hybrid Approach

Hybrid approaches are the combination of several previous approaches. For instance, from Koschke et. al. [18], the AST nodes are serialized in preorder traversal, a suffix tree is created for these serialized AST nodes, and the resulting maximally long AST node sequences are then cut out according to their syntactic regions. Thus, only syntactically closed sequences are still remained. Instead of comparing the AST nodes,their approach compares the tokens of the AST-nodes using a suffix tree-based algorithm.Therefore, this approach can find clones in linear time and space, which is a significant improvement to the usual AST-based approaches.

Another research work was presented by Jiang et. al.[28]. Certain characteristic vectors are computed to approximatethe structural information within the ASTs in Euclidean space. A Locality Sensitive Hashing (LSH) [29] is then used to cluster similar vectors w.r.t. Euclidean distance metrics and thus the code clones are formed.

Abstract Syntax Tree

In computer science, an abstract syntax tree (AST), or just syntax tree is a tree representation of the abstract syntactic structure of source code written in a programming language. Abstract syntax trees (ASTs) are created by parsing the source code, and then compared with each other by using the tree structure. ASTs are data structures widely used in compilers, due to their property of representing the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. The AST is used intensively during semantic analysis, where the compiler checks correct usage of the elements in a program and the language.

The AST captures the essential structure of the input data in a tree form, while omitting unnecessary syntactic details [30]. ASTs can be distinguished from the concrete syntactic trees by their omission of tree nodes to represent punctuation marks such as semi-colons to terminate statements or commas to separate function arguments. Tree nodes that represent unary productions in the grammar are omitted by ASTs. ASTs are generated along parsing by bottom-up approach.

When designing the nodes of a tree, a common design choice is made to determine the granularity of ASTs. That is, whether all constructs of the source language are represented as a different type of AST nodes, or whether some constructs of the source language are represented with a common type of AST nodesis all differentiated using a value [30].

Sequence Alignment

Sequence alignment is a method to calculate a corresponding relationship among strings by adding a space or shifting the alphabetic positions [2]. Sequence alignment was applied for the first time in bioinformatics. In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNS, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences [31].

The distinguished algorithm of sequence alignment is Needleman-Wunsch algorithm [32]. Needleman-Wunsch algorithm is the one used in bioinformatics to align protein or nucleotide sequences. The algorithm was found by Saul B. Needleman and Christian D. Wunsch in 1969. The Needleman-Wunsch algorithm is an efficient one based on dynamic programming.

High-Level Fuzzy Petri Net

Petri net theory has been proposed by Dr. Carl Petri in 1962 as his dissertation, "Kommunikation mit Automaten," [Communication with Automata]. Petri formulated the basis for a theory of communication between asynchronous components of a computer system. Petri nets are a graphical and mathematical modeling tool, which is concurrent, asynchronous, distributed, parallel, nondeterministic, and stochastic.They can be used to model and analyze various systems [34].

Therefore, scholars or researchers one after another conduct their researches with extended Petri net theory, such as colored Petri net [35], timed Petri net [36], fuzzy Petri net [37], high-level fuzzy Petri net [38]-[43], and so on. This paper adopts HLFPN to make a decision on the code plagiarism.

Definitions

The basic definitions and fuzzy reasoning approach are presented as follows:

- **Definition 1:** The HLFPN is defined as an eight-tuple *HLFPN* = ($P, T, F, C, V, \alpha, \beta, \delta$), where

$P = \{p_1, p_2, p_3, \ldots, p_k\}$	A finite set of places.
$T = \{t_1, t_2, t_3, \ldots, t_l\}$	A finite set of transitions. $P \cup T \neq \emptyset$
$F \subseteq (P \times T) \cup (T \times P)$	Called the flow relation and is also a finite set of arcs, each representing the fuzzy set (i.e. fuzzy term) for an antecedent or a consequent; where the positive arcs (i.e. THEN parts) are denoted by $\circ\!\!\rightarrow$.
$C = \{X, Y, Z\}$	A finite set of linguistic variables, e.g. X, Y, and Z, where $X = \{x_1, x_2\, x_3 \ldots, x_n\}$, $Y = \{y_1, y_2\, y_3 \ldots, y_m\}$, and $Z = \{z_1, z_2\, z_3 \ldots, z_q\}$.
$V = \{v_1, v_2\, v_3 \ldots, v_4\}$	A finite set of fuzzy truth values known as the fuzzy relational matrix between the antecedent and the consequent of a rule.
$\alpha : P \rightarrow C$	An association function, mapping from places to linguistic variables. $\alpha(p_i) = c_i$, i = 1, ..., I, where $C = \{c_i\}$is a set of linguistic variables in the knowledge base (KB), and I is the number of linguistic variables in the KB.
$\beta : F \rightarrow [0, 1]$	An association function, mapping from the flow relations to the fuzzy truth values between zero and one.
$\delta : T \rightarrow V$	An association function, mapping from transitions to fuzzy relational matrices.

- **Definition 2 (*Input and Output Functions*):**

$I(t) = \{p \in P \mid (p,t) \in F\}$	A set of the input places of transition t.
$I(p) = \{t \in T \mid (t,p) \in F\}$	A set of the input transitions of place p.
$O(t) = \{p \in P \mid (t,p) \in F\}$	A set of the output places of transition t.
$O(p) = \{t \in T \mid (p,t) \in F\}$	A set of the output transitions of place p.

- **Definition 3 (*Negation*):**

In the IF-THEN-ELSE rule, the ELSE part is denoted by a negation arc $\circ\!\!\rightarrow$, and the fuzzy set in the antecedent (i.e., IF part) must be complemented and denoted by $\neg$, i.e. the negated fuzzy set = 1-the fuzzy set in the antecedent.

- **Definition 4 (*Membership Function*):**

Mem(p): $P \rightarrow [0,1]$, which assigns to each place a real value *Mem*(p) = *DOM*($\alpha(p)$), where *DOM* represents the degree of membership in the associated proposition, and data tokens are available in *P.*

- **Definition 5 (*Max-Min Compositional Rule*):**

In HLFPN, $\forall$ transition t, $V(t\ )$= min(fuzzy sets in $I(t)$); and $\forall$ place p, V(p) = max(fuzzy sets in $I(p)$). This rule is denoted by $\circ$.

- **Definition 6 (*Input Place, Hidden Place, and Output Place*):**

In HLFPN, $\forall$ place $p_i \in P$, if $\forall$ $t_j \in T$, $p_i \notin O(t_j)$, then p_i is called input place (*IP*); if $\forall$ $t_j \in T$, $p_i \notin I(t_j)$, then p_i is called output place (*OP*); else, p_i is called hidden place.

Fuzzy Reasoning

In the fuzzy reasoning method presented in [43], fuzzy production rules are used. Mamdani's fuzzy implication rule type [44] is applied throughout this paper. In general, a fuzzy production rule describes fuzzy relationship between the antecedent and the consequent. Let R be a set of fuzzy production rules, where $R = \{R_1, R_2, \ldots, R_n\}$. The general form of the ith fuzzy production rule $R_i$is shown as follows:

R_i: IF d_j(X is A), THEN d_k(Y is B); ELSE, d_w(Z is C)…(V).

where "X is A", "Y is B" and "Z is C" are propositions; X is called the input linguistic variable; Y and Z are called the output linguistic variables, respectively; A is called the input fuzzy set; B and C are called the output fuzzy sets, respectively; the fuzzy truth values of the propositions "X is A", "Y is B" and "Z is C" are restricted to [0, 1]; "X is A" is the antecedent of fuzzy production rule R_i, "Y is B" and "Z is C" are the consequents of fuzzy production rule R_i. Let V represent the fuzzy relational matrix between the antecedent and the consequent of a fuzzy production rule.

Example 1:

Let us consider the fuzzy production rule R_1 shown as follows:

R_1: IF it (X_1) is hot (A_1) AND the sky (X_2) is cloudy (A_2), THEN the humidity (Y) is high (B).

Based on the transformation procedure presented in [41], we can transform the above fuzzy production rule R_1 into the following first-order logic form:

R_1': IF X_1(A_1) AND X_2(A_2), THEN Y(B).

Then, the HLFPN model is shown in Figure 1.

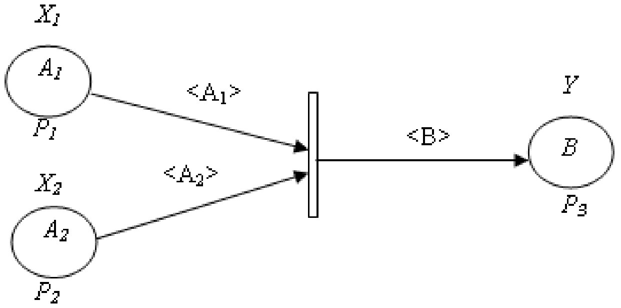


Fig 1: HLFPN for Example 1

Assume that the fuzzy sets A_1, A_2 and B are shown as follows:

$$A_1=\frac{0.24}{a_{11}}+\frac{0.55}{a_{12}}+\frac{0.25}{a_{13}}$$

$$A_2=\frac{0.13}{a_{21}}+\frac{0.78}{a_{22}}+\frac{0.42}{a_{23}}$$

$$B=\frac{0.30}{b_1}+\frac{0.72}{b_2}+\frac{0.21}{b_3}$$

By the cylindrical extension operations [45], we can obtain the antecedent fuzzy set A, shown as follows:

$$A=A_1\times A_2=(0.24\quad 0.55\quad 0.25)^T\wedge(0.13\quad 0.78\quad 0.42)$$

$$=\begin{vmatrix} 0.13 & 0.24 & 0.24 \\ 0.13 & 0.55 & 0.42 \\ 0.13 & 0.25 & 0.25 \end{vmatrix}$$

Then, the fuzzy relational matrices $V_1(t_1)$, $V_2(t_2)$ and $V_3(t_3)$ between the antecedent and consequent of fuzzy production rule R_1 can be obtained, shown as follows:

$$V_1(t_1)=\begin{vmatrix} 0.13 & 0.24 & 0.24 \\ 0.13 & 0.30 & 0.30 \\ 0.13 & 0.25 & 0.25 \end{vmatrix}\in A\times B\times b_1$$

$$V_2(t_2)=\begin{vmatrix} 0.13 & 0.24 & 0.24 \\ 0.13 & 0.55 & 0.42 \\ 0.13 & 0.25 & 0.25 \end{vmatrix}\in A\times B\times b_2$$

$$V_3(t_3)=\begin{vmatrix} 0.13 & 0.21 & 0.21 \\ 0.13 & 0.21 & 0.21 \\ 0.13 & 0.21 & 0.21 \end{vmatrix}\in A\times B\times b_3$$

The most widely used fuzzy reasoning method is the max–min composition inference [46]. Assume that the input fuzzy sets A_1' and A_2' are shown as follows:

$$A'_1=\frac{0.10}{a_{11}}+\frac{0.82}{a_{12}}+\frac{0.33}{a_{13}}$$

$$A'_2=\frac{0.28}{a_{21}}+\frac{0.87}{a_{22}}+\frac{0.49}{a_{23}}$$

Then, we can get

$$A'_1\circ V_1(t_1)=(0.10\quad 0.82\quad 0.33)\circ V_1(t_1)=(0.13\quad 0.30\quad 0.30)$$

$$A'_1\circ V_2(t_1)=(0.10\quad 0.82\quad 0.33)\circ V_2(t_1)=(0.13\quad 0.55\quad 0.42)$$

$$A'_1\circ V_3(t_1)=(0.10\quad 0.82\quad 0.33)\circ V_3(t_1)=(0.13\quad 0.21\quad 0.21)$$

Finally, we can obtain

$$B'=(0.28\quad 0.87\quad 0.49)\circ\begin{vmatrix} 0.13 & 0.13 & 0.13 \\ 0.30 & 0.55 & 0.21 \\ 0.30 & 0.42 & 0.21 \end{vmatrix}$$

$$=(0.30\quad 0.55\quad 0.21)$$

$$=\frac{0.30}{b_1}+\frac{0.55}{b_2}+\frac{0.21}{b_3}$$

The above description is the fuzzy reasoning process of HLFPN.

Fuzzy Reasoning Algorithm

In this sub-section, we briefly review the fuzzy reasoning algorithm (FRA) [37] to determine whether there exists or not a fuzzy relational matrix between the antecedent and the consequent of a fuzzy production rule.

INPUT: *Mem*(*p*), $\forall$ $p_i \in IP$, where *IP* denotes a set of input places.

OUTPUT: *Mem*(*p*), $\forall$ $p_i \in OP$, where *OP* denotes a set of output places.

PROCEDURE:

Step 1:	Initially, assume that only the DOMs in the propositions operating on input variables are available. Consequently, the initial marking function is shown as follows: $M(p_i) = 0$, if $p_i \notin IP$ $M(p_i)$ = the number of data tokens, if $p_i \in IP$
Step 2:	$\forall$ $t_j \in T$, compute $V(t_j) = W_a \times W_c = (w_{a_1}, w_{a_2}, \ldots, w_{a_m})^T \wedge (w_{c_1}, w_{c_2}, \ldots, w_{c_n})$, where T denotes a set of transitions; $V(t_j)$ is a fuzzy relational matrix between the antecedent and the consequent of rule t_j; $W_a = \{w_{a_1}, w_{a_2}, \ldots, w_{a_m}\}$ is a fuzzy set for the antecedent; $W_c = \{w_{c_1}, w_{c_2}, \ldots, w_{c_n}\}$ is a fuzzy set for the consequent; and each element of a fuzzy set is denoted by a fuzzy interval.
Step 3:	Input a data pattern $W_{a\text{-}input}$.
Step 4:	Fire the enabled transitions. Let t_j be any enabled transition. Then, compute: $t_j \in T$ / $\forall$ $p_k \in I(t_j)$, $M(p_k)$ = the number of data tokens. $W'_a = W_{a\text{-}input}$ $W'_c = W'_a \circ V(t_j)$ or $\neg W'_a \circ V(t_j)$, if an ELSE part is available.
Step 5:	For every output variable O, its associated membership distribution is $W'_c = \{w'_{c_i}\} = \vee w'_{c_i}$, i = 1, 2, …, I, where I is the in-degree of output variable O. Then, W'_c becomes an actual output.
Step 6:	Go back to Step 4, while $\exists t_j \in T / M(p_i) = 1$, $\forall p_i \in I(t_j)$, that is, while the enabled transitions still exist.
Step 7:	The weighted average defuzzification method is applied and the real operating value is obtained.

III. THE PROPOSED SYSTEM

The code plagiarism detection system based on AST and HLFPN is divided into three stages: AST generation, sequence alignment, and detection of code plagiarism using HLFPN. As shown in Fig. 2, we input the source code programs to the lexical analysis. Lexical analyzer deals with the large-scale constructs, such as expressions, statements, and program units [47]. A lexical analyzer is essentially a pattern matcher which attempts to find a substring in the given string of characters that matches a given character pattern. The lexical analysis process includes skipping comments and white space, inserts lexemes for user-defined names, and detects syntactic errors in tokens.

After lexical analysis, syntax analysis or parsing is used to construct parse trees for the given source codes. There are two distinct goals of syntax analysis [47]: First, the syntax analyzer must check the input source code to determine whether it is syntactically correct. The second goal is to produce a complete parse tree, or at least trace the structure of the complete parse tree, for syntactically correct input datasets.

The generated ASTs are compared with one another through the sequence alignment. The sequence alignment extracts the similarity features which become the input datasets to HLFPN and the decision output is obtained.

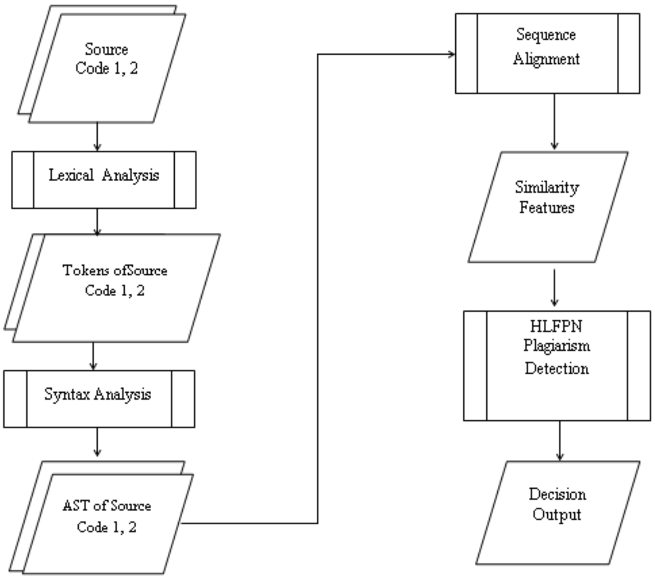


Fig 2: Flowchart of code plagiarism detection based on AST and HLFPN

Abstract Syntax Tree Generation

This paper uses an AST as the similarity detection model. ASTs can provide more accurate and comprehensive information for code plagiarism detection in terms of changing the identifier or program statement order. ANTLR (Another Tool for Language Recognition) is used to generate the syntax tree. There are two main sub-processes which are used by ANTLR to generate an AST. First, ANTLR uses a grammar file to generate the lexical and syntax analyzer. Second, the input source code is converted to an AST by using the generated lexical analyzer.The input of a parser is the phrase flow, so the AST is obtained by the parser. This process is shown in Fig. 3. For instance, the AST converted from the source code program main.c, as shown in Fig. 4, is shown in Fig. 5. The output results are the inverse of Poland expression from the source codes.

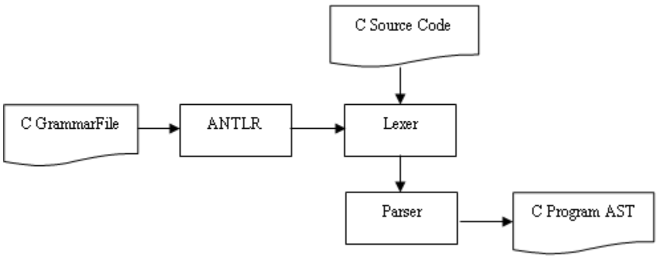


Fig 3: Flowchart of ANTLR analysis

```
//main.c

main()

{

inta,b;

}
```

Fig 4: Source code of main.c

Sequence Alignment

Sequence alignment is a method to calculate a corresponding relationship among strings by adding a space or shifting the alphabetic positions. In the proposed method, we first obtain the token sequence from the generated AST and obtain the similarity features using the concept of Needleman-Wunsch algorithm. It has a good effect on looking for the optimal matching. This algorithm is computed by using dynamic programming. Fig. 6 shows the token sequence of AST generated from Fig. 5. In our experiment, we use a scoring system for better performance as follows:

- Match score : +2
- Mismatch score : -1
- Indel score : -1

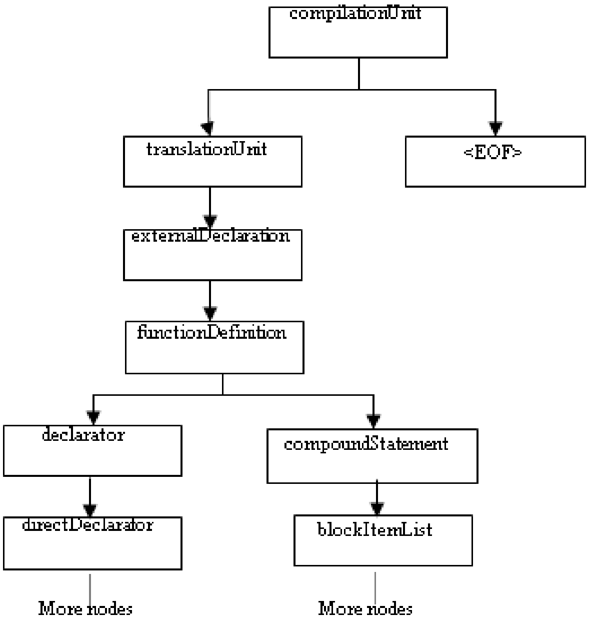
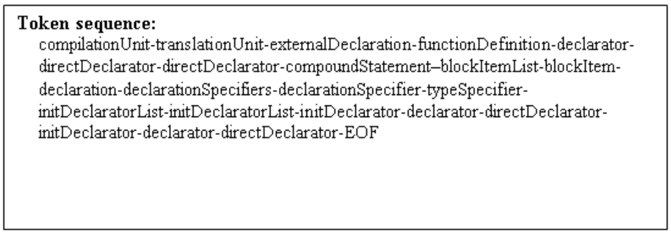


Fig 5: AST generated from main.c



Token sequence:
compilationUnit-translationUnit-externalDeclaration-functionDefinition-declarator-directDeclarator-directDeclarator-compoundStatement–blockItemList-blockItem-declaration-declarationSpecifiers-declarationSpecifier-typeSpecifier-initDeclaratorList-initDeclaratorList-initDeclarator-declarator-directDeclarator-initDeclarator-declarator-directDeclarator-EOF

Fig 6: Token sequence generated from Fig. 5

Plagiarism Detection and Similarity Feature Extraction

This sub-section explains the function and feature extraction based on HLFPN. This study uses three features to define the plagiarism decision output between two input source codes.

1) *Similarity Feature 1 : Ratio of total number of matches tothe length of the sequence*

Assume that *n* denotes the length of the sequence and n_{match} denotes the total number of matches. The ratio of total number of matches to the length of the sequence is defined as:

$$R_{match} = \frac{n_{\text{match}}}{n} \times 100\% \qquad (1)$$

2) *Similarity Feature 2 : Ratio of total number of mismatchestothe length of the sequence*

Assume that *n* denotes the length of the sequence and $n_{mismatch}$ denotes the total number of mismatches. The ratio of total number of mismatches to the length of the sequence is defined as:

$$R_{mismatch} = \frac{n_{\text{mismatch}}}{n} \times 100\% \qquad (2)$$

3) *Similarity Feature 3 : Ratioof the total number of gapstothe length of the sequence*

Assume that *n* denotes the length of the sequence and n_{gap} denotes the total number of gaps. The ratio of total number of gaps to the length of the sequence is defined as:

$$R_{gap} = \frac{n_{\text{gap}}}{n} \times 100\% \qquad (3)$$

Membership Function

In the decision method, three features are used, namely, the ratio of total number of matches to the length of the sequence (Rmatch), the ratio of total number of mismatches to the length of the sequence (Rmismatch), and the ratio of total number of gaps to the length of the sequence (Rgap). Then, three sets of similarity features membership functions are shown in Fig. 7. In addition, the plagiarism detection is also divided into three parts, namely, “Non-plagiarized”, “Undecided”, and “Plagiarized”. The membership functions for the plagiarism decision are shown in Fig. 8.

In the decision method, three features are used, namely, the ratio of total number of matches to the length of the sequence (R_{match}), the ratio of total number of mismatches to the length of the sequence ($R_{mismatch}$), and the ratio of total number of gaps to the length of the sequence (R_{gap}).The membership functions of Low, Middle, and High are defined in Table 1. The membership functions of plagiarism detection are listed in Table 2. In the analysis, the membership functions of input parameters are set between 0 and 1. Thus, the values of input parameters are converted to the values between 0 and 1.

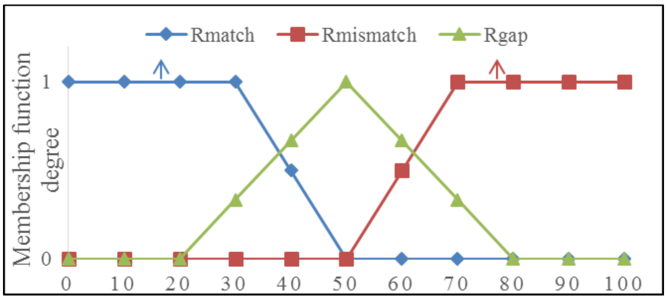


Fig 7: The type of membership functions for similarity features

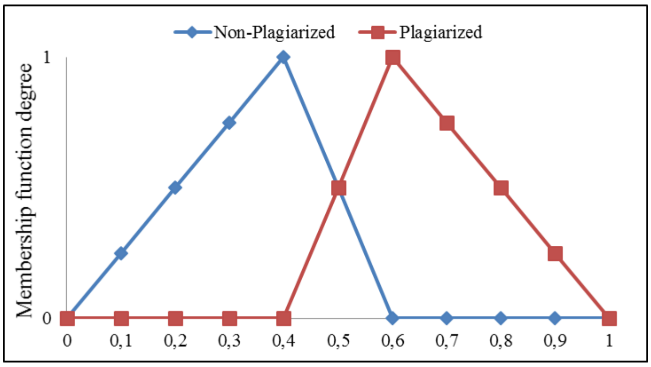


Fig 8: The type of membership functions for plagiarism decision

Table 1: Membership functions of similarity features

Input Parameter	*Low*		*Middle*			*High*	
R_{match}	10	30	20	50	80	70	90
$R_{mismatch}$	10	30	20	50	80	70	90
R_{gap}	10	30	20	50	80	70	90

Table 2: Membership functions of plagiarism decision

Non-Plagiarized		*Undecided*		*Plagiarized*	
0	0.4	0.4	0.6	0.6	1

$$\mu_H(x) = \begin{cases} 1, & x \geq 90 \\ \frac{1}{20}(x - 70), & 70 < x < 90 \\ 0, & x \leq 70 \end{cases}$$

$$\mu_M(x) = \begin{cases} 0, & x \geq 80 \\ \frac{-1}{30}(x - 80), & 50 \leq x < 80 \\ \frac{1}{30}(x - 20), & 20 < x < 50 \\ 0, & x \leq 20 \end{cases}$$

$$\mu_L(x) = \begin{cases} 0, & x \ge 30 \\ \frac{-1}{20}(x - 30), & 10 < x < 30 \\ 1, & x \le 10 \end{cases}$$

$$\mu_P(x) = \begin{cases} 1, & x \ge 1 \\ \frac{-1}{0.4}(x - 1), & 0.6 < x < 1 \\ 0, & x \le 0.6 \end{cases}$$

$$\mu_{UD}(x) = \begin{cases} 0, & x \ge 0.6 \\ \frac{-1}{0.2}(x - 0.6), & 0.4 \le x < 0.6 \\ 0, & x \le 0.4 \end{cases}$$

$$\mu_{NP}(x) = \begin{cases} 0, & x \ge 0.4 \\ \frac{-1}{0.4}(x - 0.4), & 0 < x < 0.4 \\ 1, & x \le 0 \end{cases}$$

Fuzzy Reasoning and Building HLFPN

According to the fuzzy sets and their corresponding membership functions defined in the previous sub-section, the similarity features are calculated and assigned to the fuzzifier to get the membership degrees. Therefore, an 'IF… THEN' statement is constructed in order to establish a fuzzy production rule.

Table 3. Description of parameters

Name of Parameter	*Description of Parameter*
MA	Represents the ratio of total number of matches to the length of the sequence, i.e., input place p_1.
MS	Represents the ratio of total number of mismatches to the length of the sequence, i.e., input place p_2.
G	Represents the ratio of total number of gaps to the length of the sequence, i.e., input place p_3.
D	Represents decision, i.e., output place p_4.
H, M, L	Represent high, middle, low fuzzy sets, respectively.
S, I, WK	Represent strong, intermediate, and weak fuzzy sets, respectively.
$V(t_i)$, i = 1, 2, 3	Represents the fuzzy relational matrices of *MA*, *MS*, *G*, and detection status decision.
H', M', L'	Represent high, middle, and low fuzzy sets of input values, respectively.

We configure input linguistic variables as the ratio of total number of matches (*MA*)to the length of the sequence, the ratio of total number of mismatches (*MS*)to the length of the sequence, and the ratio of total number of gaps (*G*) to the length of the sequence, with fuzzy terms: high (*H*), middle (*M*), and low (*L*). The fuzzy production rules are defined as follows:

R_1: IF *MA* is *H* THEN *D* is *S*
R_2: IF *MA* is *M* THEN *D* is *I*
R_3: IF *MA* is *L* THEN *D* is *WK*
R_4: IF *MS* is *H* THEN *D* is *WK*
R_5: IF *MS* is *M* THEN *D* is *I*
R_6: IF *MS* is *L* THEN *D* is *S*
R_7: IF *G* is *H* THEN *D* is *WK*
R_8: IF *G* is *M* THEN *D* is *I*
R_9: IF *G*is *L* THEN *D* is *S*

Based on the conversion procedure, we transform the above fuzzy production rules into the HLFPN model, as shown in Fig. 9, and the parameters are described in Table 3.

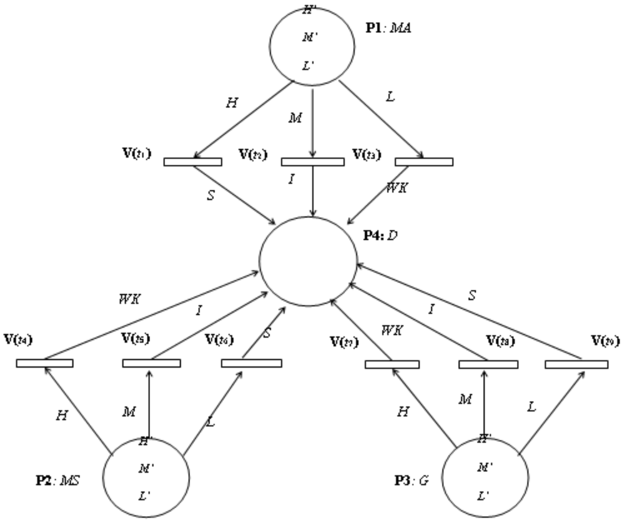


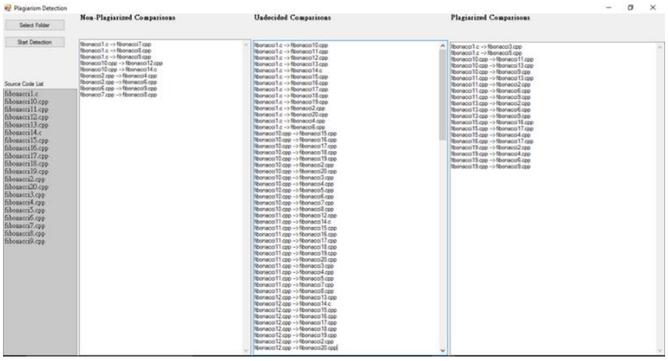
Fig 9: The HLFPN model representing nine fuzzy production rules

IV. EXPERIMENTAL RESULTS

In Section 4, we aim to experimentally evaluate the performance of our proposed system. First, we collect all the required datasets; and set up the AST using ANTLR tool and HLFPN code plagiarism detection system.Then, the experimental environment and evaluation results are discussed.

Experimental Environment

In our experiment, we have used C# programming language on Visual Studio 2015 platform. The ANTLR tool library was installed within the project on Visual Studio 2015. The user interface is shown in Fig. 10.



```
fibonacci1.c
fibonacci10.cpp
fibonacci11.cpp
fibonacci12.cpp
fibonacci13.cpp
fibonacci14.c
fibonacci15.cpp
fibonacci16.cpp
fibonacci17.cpp
fibonacci18.cpp
fibonacci19.cpp
fibonacci2.cpp
fibonacci20.cpp
fibonacci3.cpp
fibonacci4.cpp
fibonacci5.cpp
fibonacci6.cpp
fibonacci7.cpp
fibonacci8.cpp
fibonacci9.cpp
```

```
fibonacci1.c -> fibonacci7.cpp
fibonacci1.c -> fibonacci8.cpp
fibonacci1.c -> fibonacci9.cpp
fibonacci10.cpp -> fibonacci12.cpp
fibonacci10.cpp -> fibonacci14.c
fibonacci2.cpp -> fibonacci4.cpp
fibonacci2.cpp -> fibonacci6.cpp
fibonacci6.cpp -> fibonacci9.cpp
fibonacci7.cpp -> fibonacci8.cpp
```

```
fibonacci1.c -> fibonacci10.cpp
fibonacci1.c -> fibonacci11.cpp
fibonacci1.c -> fibonacci12.cpp
fibonacci1.c -> fibonacci13.cpp
fibonacci1.c -> fibonacci14.c
fibonacci1.c -> fibonacci15.cpp
fibonacci1.c -> fibonacci16.cpp
fibonacci1.c -> fibonacci17.cpp
fibonacci1.c -> fibonacci18.cpp
fibonacci1.c -> fibonacci19.cpp
fibonacci1.c -> fibonacci2.cpp
fibonacci1.c -> fibonacci20.cpp
fibonacci1.c -> fibonacci4.cpp
fibonacci1.c -> fibonacci6.cpp
fibonacci10.cpp -> fibonacci15.cpp
fibonacci10.cpp -> fibonacci16.cpp
fibonacci10.cpp -> fibonacci17.cpp
fibonacci10.cpp -> fibonacci18.cpp
fibonacci10.cpp -> fibonacci19.cpp
fibonacci10.cpp -> fibonacci2.cpp
fibonacci10.cpp -> fibonacci20.cpp
fibonacci10.cpp -> fibonacci3.cpp
fibonacci10.cpp -> fibonacci4.cpp
fibonacci10.cpp -> fibonacci5.cpp
fibonacci10.cpp -> fibonacci6.cpp
fibonacci10.cpp -> fibonacci7.cpp
fibonacci10.cpp -> fibonacci8.cpp
fibonacci11.cpp -> fibonacci12.cpp
fibonacci11.cpp -> fibonacci14.c
fibonacci11.cpp -> fibonacci15.cpp
fibonacci11.cpp -> fibonacci16.cpp
fibonacci11.cpp -> fibonacci17.cpp
fibonacci11.cpp -> fibonacci18.cpp
fibonacci11.cpp -> fibonacci19.cpp
fibonacci11.cpp -> fibonacci20.cpp
fibonacci11.cpp -> fibonacci3.cpp
fibonacci11.cpp -> fibonacci4.cpp
fibonacci11.cpp -> fibonacci5.cpp
fibonacci11.cpp -> fibonacci7.cpp
fibonacci11.cpp -> fibonacci8.cpp
fibonacci12.cpp -> fibonacci13.cpp
fibonacci12.cpp -> fibonacci14.c
fibonacci12.cpp -> fibonacci15.cpp
fibonacci12.cpp -> fibonacci16.cpp
fibonacci12.cpp -> fibonacci17.cpp
fibonacci12.cpp -> fibonacci18.cpp
fibonacci12.cpp -> fibonacci19.cpp
fibonacci12.cpp -> fibonacci2.cpp
fibonacci12.cpp -> fibonacci20.cpp
```

```
fibonacci1.c -> fibonacci3.cpp
fibonacci1.c -> fibonacci5.cpp
fibonacci10.cpp -> fibonacci11.cpp
fibonacci10.cpp -> fibonacci13.cpp
fibonacci10.cpp -> fibonacci9.cpp
fibonacci11.cpp -> fibonacci13.cpp
fibonacci11.cpp -> fibonacci2.cpp
fibonacci11.cpp -> fibonacci6.cpp
fibonacci11.cpp -> fibonacci9.cpp
fibonacci13.cpp -> fibonacci2.cpp
fibonacci13.cpp -> fibonacci6.cpp
fibonacci13.cpp -> fibonacci9.cpp
fibonacci15.cpp -> fibonacci16.cpp
fibonacci15.cpp -> fibonacci17.cpp
fibonacci15.cpp -> fibonacci4.cpp
fibonacci16.cpp -> fibonacci17.cpp
fibonacci19.cpp -> fibonacci2.cpp
fibonacci19.cpp -> fibonacci4.cpp
fibonacci19.cpp -> fibonacci6.cpp
fibonacci19.cpp -> fibonacci9.cpp
```

Fig 10: User interface

Experimental Procedure

In this sub-section, we present the experimental procedure of our research as follows:

1. *Determine all the sample exercises which have to be performed by students.*

We select four sample exercises which have to be performed by students in the class using *C* programming language. The number of students in the class is 20. Thus, each student needs to perform five sample exercises presented in Table 4. Then, we collect all the exercises done by the students to detect the code plagiarism.

Table 4. Sample exercises

Exercise	*Description*
E1	Simple Calculator
E2	Merge Sort
E3	Binary Search
E4	Fibonacci Number
E5	HanoiTower

2. *Detect the code plagiarism among all the exercises.*

Before the code plagiarism detection is performed, we group each sample exercise as one folder. The code plagiarism detection system is developed to compare any two students' exercises within the same folder and yield the decision output.

3. *Discuss the decision on students' work.*

After the decision output is obtained, we make a discussion with the student about his/her algorithm and make a decision, either plagiarized or non-plagiarized.

4. *Measure the precision rate of our proposed system.*

Main Results

As tabulated in Table 5, this experiment evaluates 5 sample exercises. Each exercise contains 20 source codes created by 20 students. Total number of source codes is 100 with 950 comparisons of any two source codes within the same folder. After our approach was performed, we obtained 591 *Non-plagiarized*, 243 *Undecided* and 116 *Plagiarized* detection outputs.

Table 5. Information of source codes

Exercise	*No. of Source Codes*	*No. of Comparisons*	*Non-plagiarized*	*Undecided*	*Plagiarized*
E1	20	190	122	35	33
E2	20	190	117	50	23
E3	20	190	119	55	16
E4	20	190	118	52	20
E5	20	190	115	51	24
Total	***100***	***950***	***591***	***243***	***116***

In Table 5, we can see that our approach yields more Undecided comparisons than Non-plagiarized comparisons. This occurs because all source codes have a similar algorithm to perform an exercise.

In order to perform a fair and comparative evaluation, we compare our proposed system with AST-based code plagiarism detection system without HLFPN using precision calculation. Precision is the ratio of the number of correctly detected code plagiarisms to total number of correctly and incorrectly detected code plagiarisms in source code comparisons. The formula for precision is shown in Equation (10).

$$Precision = \frac{Correct\ detection\ outputs}{Correct\ detection\ outputs + Incorrect\ detection\ outputs} \text{ x } 100\% \quad (10)$$

The performance evaluation results are shown in Table 6. The larger the evaluated values are, the better the code plagiarism detection will become.

Table 6. Performance analysis

Exercise	*Precision, %*		
	Without HLFPN	*With HLFPN*	*Increment*
E1	86.73	94.74	8.01
E2	77.89	90.00	12.11
E3	92.63	95.78	3.15
E4	80.52	93.15	12.63
E5	87.37	92.10	4.73
Average	***85.03***	***93.15***	***8.12***

Based on the performance analysis results in Table 6, we can see that on average the precision for the approach AST without HLFPN can only achieve 85.03% correctly. However, after integrating the AST with HLFPN, we can achieve the average precision as high as 93.15%. It has increased the precision by 8.12%. The experimental results indicate that the proposed approach can achieve the reliable improvement.

V. CONCLUSIONS

This paper has proposed a code plagiarism detection system using the HLFPN model based on ASTs. To do so, we first need to construct the AST and generate the token sequence. The token sequences from two ASTs are used to obtain the similarity features which are adopted as input datasets to the HLFPN model. The contributions of this study are presented as follows:

1. By using an HLFPN model based on ASTs, our system is proved to detect the source code plagiarism which cannot be defeated by comments modification, renaming identifiers, reordering the block of code, reordering the sentences within a block, changes of operator or operand sequence in an expression, changes of data type, splitting of an expression, replacement of control structure by equivalence control structure, increase of the redundancy of statements or variables, and combination of all the above scenarios.

2. Improving the performance of previous AST approach without HLFPN can better detect the code plagiarism.

3. Due to the "Undecided" output, it gives the teacher an opportunity to discuss with the students about their source codes. Thus, it prevents the teacher from directly judging a student as a plagiarist.

From the experimental results, we know that although our approach can detect the code plagiarism; it still yields more Undecided outputs. It occurred due to the simplicity of the sample exercises and the similarity of the source codes. In the future, we will do more analyses of similarity features and scoring system of sequence alignment to deal with the simple exercises with optimization and efficiency of the code plagiarism detection method, which tackle more programming languages.

Acknowledgments

The authors are very grateful to the anonymous reviewers for their constructive comments which have improved the quality of this paper.Also, this work was supported by the Ministry of Science and Technology, Taiwan, under grant MOST 107-2221-E-845-002-MY3.

References

[1]. S. Butakov, M. Kim, and S. Kim, "Low RAM footprint algorithm for small scale plagiarism detection projects," Procs. of the International Conference on Information Science and Applications (ICISA), pp. 1-2, May 2012.

[2]. H. Kikuchi, T. Gooto, M. Wakatsuki, and T. Nishino, "A source code plagiarism detecting method using alignment syntax tree elements," Procs. of the 15th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 1-6, Jun. 2014.

[3]. S. Nadelson, "Academic misconduct by university students: Faculty perceptions and responses," Plagiary, vol. 2, no. 2, pp. 1-10, 2007.

[4]. F. Culwin and T. Lancaster, "Plagiarism issues for higher education," Procs. of VINE, vol. 31, no. 2, pp. 36-41, 2001.

[5]. J. Wilkinson, "Staff and student perceptions of plagiarism and cheating," International Journal of Teaching and Learning in Higher Education, vol. 20, no. 2, pp. 98-105, 2009.

[6]. M. Paris, "Source code and text plagiarism detection strategies," Procs. of the 4th Annual LTSN-ICS Conference, pp. 74-78, 2003.

[7]. M. Dick, et. al., "Addressing student cheating: Definitions and solutions," Procs. of Innovation and Technology in Computer Science Education," pp. 172-184, Jun. 2002.

[8]. M. Joy, G. Cosma, J. Y. Yau, and J. Sinclair, "Source code plagiarism-A student perspective," IEEE Transactions on Education, vol. 54, no. 1, Feb. 2011.

[9]. E. Jones, "Metrics based plagiarism monitoring," Journal of Computing Sciences in Colleges, vol. 16, no. 4, pp. 253-261, 2001.

[10]. J. Zhao, K. Xia, Y. Fu, and B. Cui, "An AST-based code plagiarism detection algorithm," Procs. of the 10th International Conference on Broadband and Wireless Computing, Communications, and Applications (BWCCA), pp. 178-182, Nov. 2015.

[11]. C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's UniversityTechnical Report No. 2007-541, pp. 1-109, Sept. 2007.

[12]. B. Baker, "On finding duplication and near-duplication in large software systems," Procs. of the Second Working Conference on Reverse Engineering, pp. 86-95, Jul. 1995.

[13]. J. Johnson, "Substring matching for clone detection and change tracking," Procs. of the 10th International Conference on Software Maintenance, pp. 120-126, 1994.

[14]. T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654-670, Jul. 2002.

[15]. Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," Procs. of the 6th Conference on Symposium on Operation Systems Design & Implementation, pp. 289-302, Dec. 2004.

[16]. L. Prechelt, G. Malpohl, and M. Phillipsen, "Finding plagiarisms among a set of programs with JPlag," Journal of Universal Computer Science, vol. 8, no. 11, pp. 1016-1038, Apr.2002.

[17]. S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," SIGMOD ACM, pp. 76-85, Jun. 2003.

[18]. R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," Procs. of IEEE 13th Working Conference on Reverse Engineering (WCRD), pp. 253-262, Oct. 2006.

[19]. L. P. Zhang and D. S. Liu, "AST-based multi-language plagiarism detection method," Procs. of IEEE 4th International Conference on Software Engineering and Service Science (ICSESS), pp. 738-742 , May 2013.

[20]. J. Feng, B. Cui, and K. Xia, "A code comparison algorithm based on AST for plagiarism detection," Procs. of the 4th International Conference on Emerging Intelligent Data and Web Technologies (EIDWT), pp. 393-397, Sept. 2013.

[21]. L. P. Zhang, D. S. Liu, Y. Li, and M. Zhong, "AST-based Plagiarism Detection Method," Procs. of the International Workshop on Internet of Things' Technology and Innovative Application Design (IOT Workshop), pp. 611-618, Apr. 2012.

[22]. S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo,"Comparison and evaluation of clone detection tools,"IEEE Transactions on Software Engineering, vol. 31, no. 10, pp.804–818, Aug. 2007.

[23]. Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto, "Incremental code clone detection: A PDG-based approach," Procs. of the 18th IEEE Working Conference on Reverse Engineering, pp. 3-12, Oct. 2011.

[24]. M. Balint, T. Girba, and R. Marinescu, "How developers copy," Procs. of the 14th IEEE International Conference on Program Comprehension, pp. 56–68, Jun. 2006.

[25]. R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," Procs. of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages, pp. 155–169, Jan. 2000.

[26]. J. Krinke, "Identifying similar code with program dependence graphs," Procs. of the 8th Working Conference on Reverse Engineering, pp. 301–309, Oct. 2001.

[27]. Christopher Venters, Cassandra Groen, Lisa D. McNair, and Marie C. Paretti,"Using writing assignments to improve learning in Statics: A mixed methods study", The International Journal of Engineering Education, vol. 34, no. 1, pp. 119-131, Feb. 2018.

[28]. L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," Procs. of the 29th International Conference on Software Engineering (ICSE'07), pp. 96-105, May 2007.

[29]. M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," Procs. of the 20th annual symposium on computational geometry (SoGG'04), pp. 253-262, Jun. 2004.

[30]. J. Jones. 2016. Abstract Syntax Tree Implementation Idioms [Online]. University of Alabama. Available: http://www.hillside.net/plop/plop20013/Papers/Jones-ImplementingASTs.pdf.

[31]. D. W. Mount, Sequence and Genome Analysis, Cold Spring Harbor Laboratory Press, 2002.

[32]. Wiharyanto Oktiawan, Mochtar Hadiwidodo, and Purwono,"Enhancement student understanding through the development of lab module based on constructivistic", The International Journal of Engineering Education, vol. 1, no. 1, pp. 1-5,Jan. 2016.

[33]. T. Akutsu, Mathematical Models and Algorithms in Bioinformatics, Kyoritsu Shuppan, 2007.

[34]. T. Murata, "Petri nets: Properties, analysis and applications," Proceedings of IEEE, vol. 77, no. 4, pp. 541-580, Aug. 1989.

[35]. R. Robidoux, H.P. Xu, L.D. Xing, and M.C. Zhou, "Automated modeling of dynamic reliability block diagrams using colored Petri nets," IEEE Transactions on Systems, Man, Cybernetics-Part A: Systems and Humans, vol. 40, no. 2, pp. 337–351, Nov. 2010.

[36]. H. Ogata and Y. Yano, "Knowledge awareness map for computer-supported ubiquitous language-learning," Procs. of the 2nd IEEE International Workshop on Wireless and Mobile Technologies in Education, pp. 19–25, Mar. 2004.

[37]. Ari Wibisono, Wisnu Jatmiko, Hanief Arief Wisesa, Benny Hardjono, and Petrus Mursanto, "Traffic big data prediction and visualization using Fast Incremental Model Tress-Drift Detection (FIMT-DD)," Knowledge-Based Systems, vol. 93, pp. 33–46, Feb. 2016.

[38]. Victor R.L. Shen and Cheng-Ying Yang, "An intelligent multiagent tutoring system inartificial intelligence", The International Journal of Engineering Education, vol. 27, no. 2, pp. 248-256, Apr. 2011.

[39]. Massimo Bartoletti, Tiziana Cimoli, and G. Michele Pinna, "Lending Petri nets," Science of Computer Programming, vol. 112, no. 1, pp. 75–101,Nov.2015.

[40]. Kaile Zhou, and Shanlin Yang, "Exploring the uniform effect of FCM clustering: A data distribution perspective," Knowledge-Based Systems, vol. 96,pp. 76–83, Mar. 2016.

[41]. V. R. L. Shen, H. Y. Lai, and A. F. Lai, "The implementation of a smartphone-based fall detection system using a high-level fuzzy Petri net," Applied Soft Computing, vol. 26, no. 1, pp. 390-400, Jan. 2015.

[42]. V. R. L. Shen, "Knowledge representation using high-level fuzzy Petri nets," IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, vol. 36, no. 6, pp. 2120-2127, Oct. 2006.

[43]. V. R. L. Shen, "Reinforcement learning for high-level fuzzy Petri nets," IEEE Transactions on Systems, Man, and Cybernetics-Part B: Cybernetics, vol. 33, no. 2, pp. 351-362, Mar. 2003.

[44]. E. H. Mamdani, "Application of fuzzy logic to approximate reasoning using linguistic systems," IEEE Transactions on Computers, vol. 26, no. 12, pp. 1182–1191, Dec. 1977.

[45]. V. R. L. Shen, "Correctness in hierarchical knowledge-based requirements," IEEE Transactions on Systems, Man, and Cybernetics-Part B: Cybernetics, vol. 30, no. 4, pp. 625-631, Aug. 2000.

[46]. V. R. L. Shen, Y. S. Chang, and T. T. Y. Juang, "Supervised and unsupervised learning by using Petri nets," IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, vol. 40, no. 2, pp. 363-375, Mar. 2010.

[47]. R. W. Sebesta, Concepts of Programming Languages, New Jersey: Pearson Education, 2012.