2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing

Automatic Source Code Plagiarism Detection

Cynthia Kustanto, Inggriani Liem
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
cynthia.kustanto@yahoo.com, inge@if.itb.ac.id

Abstract— **Plagiarism is one form of academic dishonesty, which is often done by students in programming classes. In a large class, detecting plagiarism manually is both difficult and time-consuming、especially due to the numerous modifications of the source code to conceal the cheating.**

We designed and developed Deimos, a prototype of a source code plagiarism detector, which can be extended to handle other programming languages, simply by implementing new scanners and parsers.

Deimos works in two steps: (1) parsing source code and transforming it into tokens, and then (2) comparing each pair of token strings obtained in the first step using Running Karp-Rabin Greedy String Tiling algorithm.

Instructor can access Deimos via a web application interface that receives input parameters, triggers a background process, and displays the result. The web interface offers user friendliness while the background process prevents timeout and reduces bandwidth consumption. This approach was chosen since Deimos is intended to be used for processing more than 100 source code. The web application was implemented using PHP, while Java was used to implement the backend application, which is responsible for the background process.

Unit test, functional test, and nonfunctional test has been conducted. Detection time is 1 hour for processing 100 samples of beginner's source code taken from real assignment of our programming class where the average length of source code is 150 lines. This code similarity detector could also be used for other pedagogical tools, such as autograder, which checks consistency of source code based on a template or solution.

plagiarism detection, source code plagiarism

I. INTRODUCTION

One form of dishonesty which occurs frequently in the academic environment is plagiarism. Plagiarism is the act of imitating or copying or using somebody else's creation or idea without permission and presenting it as one's own [1]. One example of its practice in programming courses is when a student submits work of which part was copied from another student's work, assuming that instructors will not find out about the truth. There are various reasons why students plagiarize, most of which are related to laziness or defects in academic ability [2]. In spite of the simple programming tasks, it is important for students to try doing the assignments of basic programming courses by themselves.

Source code plagiarism in programming courses has characteristics which are different from other kinds of plagiarism [3]. It is easy to do, but difficult to detect [4]. Students in a programming course work on the same problem in the same environment, so there is large possibility that their programs will be similar. It is also common that students make certain modifications to the source code without changing the program's output. A plagiarized source code can also be a combination of several segments copied from different sources, or a combination of original work and others' works. These modifications increase the difficulty in recognizing plagiarism.

Source code modification can be classified into either lexical change or structural change [2]. Lexical change is simple modification that can be done with a text editor without any programming knowledge, such as addition/removal of comments and modification of identifiers. Lexical change can be easily detected. On the other hand, one needs certain level of programming knowledge to make structural changes. This kind of modification makes detection even more difficult. Examples of structural changes are changing iterations, changing conditional statements, changing the order of statements, changing procedure to function and vice versa, changing procedure call within the body of the procedure and vice versa, adding statements that will not affect the output of the program, changing expression to its equivalent form, etc.

Another characteristic of beginner programming course is the large number of students, which increases the difficulty of plagiarism detection. In our case, we have to handle 100 to 350 students in parallel classes as one batch of students. Every week, students submit assignments on which instructors must give feedback. The combination of a large class size along with a high frequency of assignments makes it both ineffective and inefficient to detect plagiarism manually. By using an automated detector, we hope that plagiarism detection for multiple programs can be performed more efficiently. The automatic detector also must not be cheated by modifications that were done to conceal plagiarism act. The detector also must be able to distinguish instances of plagiarism cases from accidental

978-0-7695-3642-2/09 $25.00 © 2009 IEEE
DOI 10.1109/SNPD.2009.62

similarities, so that detection errors can be minimized. The detector must also be designed to be able to process source code in any programming language, since our programming courses use at least three languages.

Grading a large number of source code each week gives us motivation to set up an autograder system. We aim to give students appropriate feedback so that students will not make the same mistakes in the next assignment. This paper presents the plagiarism detection feature as a part of the autograder system.

II. Related Work

Various approaches have been proposed for detecting source code plagiarism [5]. Each of these approaches focuses on certain characteristics of code plagiarism. For example, there are approaches which are designed mainly to compare source codes written in different programming languages. There are also approaches which are designed to handle complicated code modification but require longer detection time compared to common approaches. One of the approaches that we considered suitable for detecting plagiarism in programming course is the structure-based method, which mostly use tokenization and string matching algorithm to measure similarity. Some of existing plagiarism detectors that employ such structure-based methods are **Plague**[6], **YAP**[7], and **JPlag**[8].

Plague is one of the earliest structure-based detectors. Plague works in several steps. First, structure profiles of each source code are created. Then, those structure profiles are compared using Heckel algorithm. Suggested by Paul Heckel, the algorithm is designed to handle text files. **Plague**'s detection results are returned in the form of lists. By using a corresponding interpreter, the results can be processed further to make it easier to comprehend for common users. **Plague** is able to detect plagiarism for source code written in C. **Plague** uses some UNIX utility, so there are some portability problems.

Next, **YAP** was developed based on **Plague** with some enhancements. The first version was created by Michael Wise in 1992. Then it was optimized into **YAP2**. The final version **YAP3** was created in 1996, which can also be used to detect text plagiarism for [9]. All three versions of **YAP** have two phases in their processes. The first phase is the generation phase, where a token file is created for each source code. A token file commonly contains 100-150 tokens for small assignments and 400-700 tokens for large assignments. The second phase is comparison of every token files. The result of each comparison is a value called percent match, a value between 0 and 100. Moreover, user can specify a minimum value. If the percent match of a pair of token files is larger than this minimum value, then the corresponding pair will be judged as a case of suspected plagiarism. **YAP**'s detection result is presented in the form of a text file.

The main difference of the three versions of **YAP** is the algorithm used during the comparison phase. **YAP1** uses LCS algorithm. This algorithm's weakness is that the order of tokens affects the comparison result, so plagiarism can easily be concealed by modifying statement order. To solve this statement order problem, **YAP2** uses Heckel algorithm. However, this algorithm does not prioritize longer substrings. Identical substrings being identified by this algorithm are mostly in the form of several short substrings scattered over the source code. **YAP3** uses Running Karp-Rabin Greedy String Tiling algorithm as the comparison algorithm to handle the deficiencies presented by **YAP** and **YAP2**.

JPlag is a system that can be used to detect plagiarism for source code written in Java, C, C++ and Scheme. It is available as a free web service. Its input is a directory containing programs that will be detected. Every source code in the directory are parsed and transformed to token strings. These token strings will be compared to each other using Running Karp-Rabin Greedy String Tiling algorithm. **JPlag**'s detection result is displayed as a group of HTML files that can be opened using a standard browser. Detection statistics, similarity distribution, and pairs of programs suspected as plagiarism instances are shown on the main page. The user can also choose a certain pair of program to be shown side-by-side. Similar segments of the code will be marked with different font colors. With this kind of user interface, the user can easily analyze the results.

YAP3 and **JPlag** apply Running Karp-Rabin Greedy String Tiling (RKR-GST) as the comparison algorithm. This algorithm was suggested by Michael Wise for computing similarity level of two strings [10,11]. The objective of this algorithm is to find matching substrings from two token strings without overlapping by maximizing the number of marked tokens. RKR-GST algorithm has a parameter called minimum-match-length, which can be used to represent the sensitivity of detection. All matching substrings of which lengths are shorter than the minimum-match-length will be ignored. Furthermore, Karp-Rabin hashing technique [12] is applied to reduce the algorithm complexity. Hash values for every substring are counted and then compared. If a hash value associated to a pair of substrings has the same value, then there is probability that the corresponding pair of substrings are similar. Although the worst complexity of RKR-GST is $O(n^3)$, it has been tested that the occurrence of this worst case condition is impossible with the application of the Karp-Rabin technique. In practice, the estimation of complexity will be between OS(n) to $O(n^2)$.

III. Our Approach

Based on the characteristics of our programming course for beginners, **Deimos** must be able to detect plagiarism for source code written in two programming languages with different paradigms, i.e. LISP and Pascal programming language. **Deimos** must also be extendable so that it can detect plagiarism for source code written in other programming languages. To ensure this capability, we

planned to extend **Deimos** so that it can handle source code written in C programming language.

In summary, **Deimos** has the following as main functional requirements: (a) to detect plagiarism, (b) to display detection result in a readable form, and (c) to delete detection result. Meanwhile, nonfunctional requirements of **Deimos** are described as follows: (a) efficient detection time, (b) accessibility from any computer, (c) potential for further development to handle other programming languages without rebuilding the whole application, and (d) detection sensitivity that can be set.

We adopted some aspects of existing structure-based detectors that have been described in section II. We divided the detection process in two phases: (a) tokenization and (b) comparison of every pair of token strings. Tokenization is undertaken by parsing source code into groups of tokens, called token strings. A token is a single element of a programming language, for example a reserved word or an operator. In phase (b), the token strings produced in phase (a) are compared to each other. For n number of programs in one submission, there will be n*(n-1)/2 comparisons. RKR-GST algorithm is implemented for this comparison process. RKR-GST is suitable for plagiarism detection since it prioritizes longer substrings and it is not greatly affected by the order of substrings [11].

In RKR-GST algorithm, hash values are counted for every substring. For faster hashing process, each token is represented as an integer. If the tokens are in the form of strings, hash values for every token must be counted first.

To achieve the extendable feature requirement, there must be a separated unit to handle the tokenization process for each language. While the tokenization process is language-dependent, comparison process is language independent. That is why we need to build specific parser for each programming language and a single comparison module for all languages.

Accessibility is another feature desirable for our program. A large programming class usually consists of many instructors and teaching assistants involved in the assessment process. It is important to have a detector that is easily accessed. To achieve this objective, a web interface is a good choice. It provides the means for any instructors/assistants to interact with the application from any computer, including executing detection and observing detection result. The raw detection result is stored in a database. Whenever the instructors/assistants request a display of detection result, the stored data is processed using server-side script and shown in HTML format.

In beginner programming class such as ours, the first series of problems are simple and the corresponding solutions are in the form of short programs. Thus, the modifications done to conceal plagiarism will also be simple as students have not acquired much programming knowledge yet. However, during the progression of the course, both the difficulty of the problems as well as the students' skills increases, increasing the complexity involved in plagiarism detection. Therefore, it is necessary for the instructor to set the sensitivity of detection based on the assignments' level of difficulty. To solve this issue, the parameter minimum-match-length of RKR-GST algorithm can be set by the instructor, so that the sensitivity of detection is appropriate. Deciding the most optimal minimum-match-length value is important as smaller minimum-match-length value means higher sensitivity. If there are a lot of scattered similar segments on a pair of source code, then high sensitivity will make all those segments be suspected as plagiarism instances. If the minimum-match-length is set to a larger value, then short similar segments will be ignored and assumed as accidental similarity. To determine this minimum-match-length value, the average length of source code and the level of difficulty of the programming assignment should be taken into account.

IV. DESIGN AND IMPLEMENTATION

Deimos consists of two applications. The first part is a web application which can be accessed through a web browser. The second part is a backend application, which is responsible for doing the actual detection in response to an invocation message from the front-end.

The web application acts as the user interface to interact with plagiarism detection functions. Even though we can find an efficient detection algorithm, timeout might occur when processing more than one hundred source code. We designed the process as backend application to prevent such a case from occuring. The backend application is responsible for the detection process and storing of the results in a database.

Figure 1 depicts the architectural design of **Deimos**. Tokenizer (programming language-dependent) is separated from comparator (programming language-independent). To make a detector that is capable of handling another programming language, a new tokenizer can be implemented without changing the whole architecture.

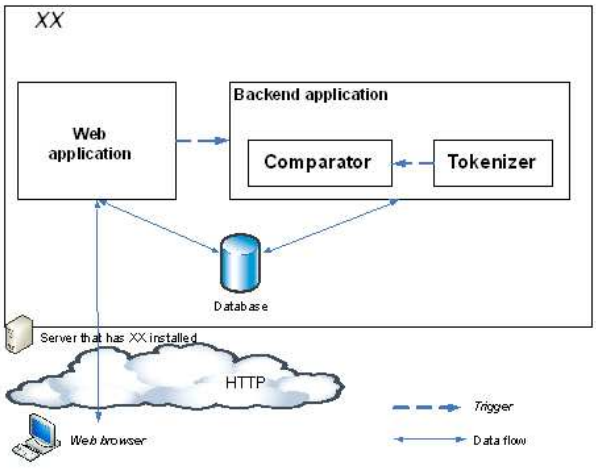

Figure 1. **Deimos** architectural design

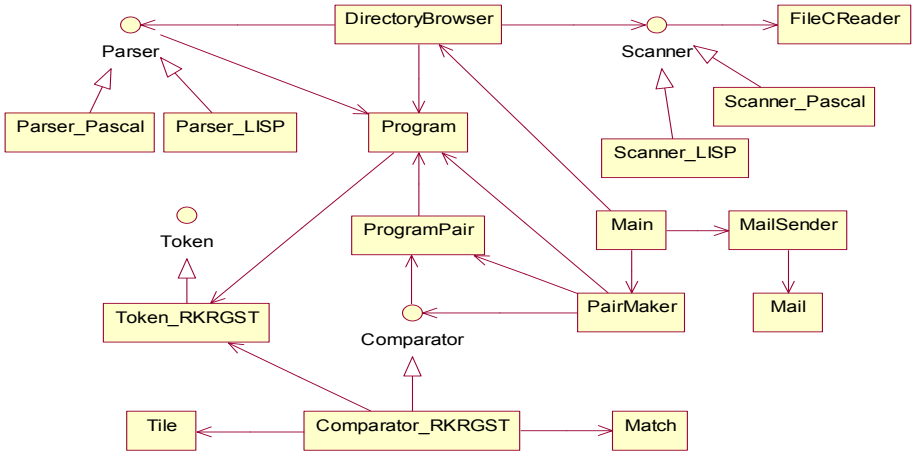

Figure 2. Class diagram of backend application

Front-end applications are implemented with object-oriented programming such as PHP, while backend applications are implemented using Java. Class diagram of the backend application can be seen at Figure 2 (The figure only shows class names and class hierarchy; it does not show the attributes and methods due to space limitation.). PHP and Java were chosen due to their portability. Java and PHP can work in various computers even with different operating systems.

The detection process follows the subsequent order. First, the tokenizer reads a set of source code in the input directory. Each source code is scanned and parsed while the integer representations of the programs' segments are stored in the database. Then, the comparator compares token strings for each pair of source code. In every comparison, a similarity value is computed and stored in the database. Similar segments of every pair are associated with specific color and their positions are stored in the database as well.

Currently, **Deimos** is able to detect plagiarism for source code written in LISP and Pascal. A scanner is not mandatory for obtaining tokens. Since the syntax of LISP is simple, it is possible to produce token strings in one pass. For LISP, we simply implemented a parser, so less detection time is needed. On the other hand, for source code written in Pascal programming language, scanning and parsing processes are done sequentially. The results of the parsing process are token strings that will be compared at the next phase.

To prove the extendability of **Deimos** for handling other programming languages, we also created a new instance of tokenizer for C language. So far, this version has only been developed to handle a subset of C instructions that is frequently used in our assignments. The steps of adding a new language feature are as follows: (a) define a token set, (b) implement a new scanner and parser, (c) register the name of the language to the interface, and (d) put a call statement in the source code of Directory Browser class. After executing these steps, **Deimos** can handle source code written in C. This functionality has been tested.

The comparator computes similarity values for every pair of source code. The input is token strings that were produced by tokenizer.

To compute similarity value, we used a formula which is adopted from YAP. The similarity formula used in **Deimos** is as the following

$$\text{Sim} = \text{t\_tiled}/\text{min\_prg\_length}*100\%. \quad (1)$$

Where Sim is the percentage value of the program pair, t_tiled is the number of similar tokens on the pair obtained from RKR-GST algorithm, and min_prg_length is the number of tokens on the shorter program from the pair. In some cases of partial plagiarism, only some segments of the original source code that were managed to be copied, so the length might be much shorter compared to the original one. For example, a student copied another student's source code, which actually hasn't been finished. The copied source code was submitted just as it is, which means that the whole source code is 100% result of plagiarism. However, the original was fixed and completed so that the length of the original source code becomes longer than the copied one. With this formula, if such case happens the user will be alerted that an instance of plagiarism has been detected in one of the pairs.

To trigger detection, the user must input system parameters (i.e. programming language of the source code, sensitivity level) and source code location through the web application. When detecting plagiarism on many files of source code, the instructor can begin the detection process and let the system operate without further input. He/she can use the computer for other activities and will receive a notification by email when the detection process finished. Even if the workstation gets disconnected, detection process in the server can continue. In situations where connection to the Internet is not reliable, this feature is particularly

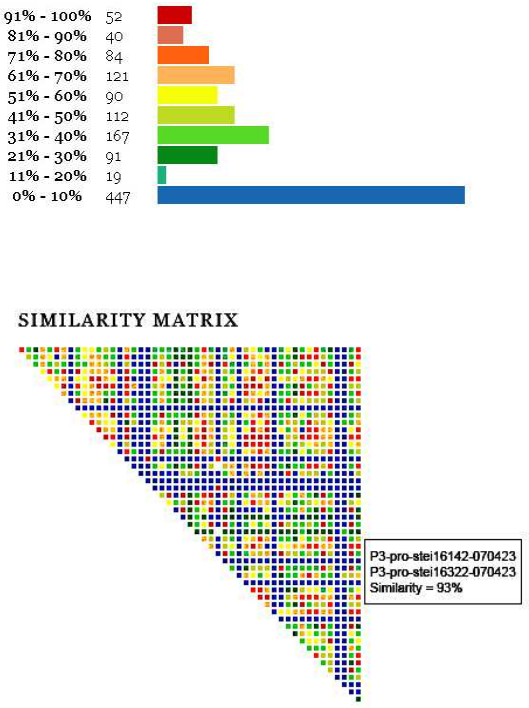

Figure 4. Global view of detection results

advantageous. However, if there is only a few files of source code that need to be processed, results can be acquired immediately.

Visualization of the detection results is available at two different levels: global view and detailed view. When the user requests a display of the detection results, the main page will show detection statistics, a similarity value distribution chart and a matrix containing similarity values for every pair of source code.

For a batch of source code (for example, 50 submissions from one assignment, see Figure 3), the instructor obtains a similarity value distribution chart and a matrix with 50x50 cells, where each cell represents a pair of source code. Range of similarity values is represented by color, which varies every 10 percent following the spectrum of RGB code. The highest range of similarity values (91%-100%) is represented by red, while lowest range of similarity values (0%-10%) is represented by blue.

The similarity matrix also acts as a panel where instructor can select and examine further the details for a pair of source code. When the user clicks on one of the pair, it will lead to a page showing both source code side-by-side with the suspected plagiarized segment marked with different colors (see Figure 4).

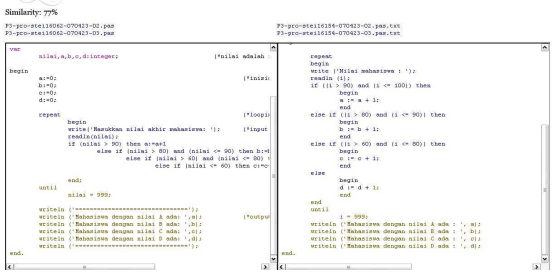

Figure 3. Detailed view of detection results

V. TESTING

Three testing methods have been undertaken: (a) unit testing, (b) functional testing, and (c) nonfunctional testing. Each class and web page passed through unit testing. Test drivers were developed for scanner, parser, and comparator unit. Functional testing was conducted based on use case scenarios *[ref to our technical documentation]* following user (instructor) tasks.

As we have mentioned in the previous sections, the matter of choosing minimum-match-length is very important. There is a large possibility of Type I error occuring when minimum match length of value 1 is chosen. As students in a programming course work on the same simple problems, it is unsurprising that many parts of the source code will be similar. With a minimum-match-length of value 1, even if there is only one similar statement among different statements, that single matter will be added to the computing of the similarity value. This will make it more difficult for the user to analyse plagiarism cases as there will be many nonplagiarized pairs of source code with high similarity values.

Meanwhile, if the minimum match length is too large compared to the length of the source code, Type II errors might occur. For example, if a student plagiarized and transformed a part of the source code into a new procedure, there will be different statements slipped inside the source code. Because the minimum-match-length was set by a large value, the detection will ignore any similar parts of the source code of which length is shorter than that value, and produce a similarity value that is smaller than expected. This is why the rough average length of source code should be considered when deciding the value minimum-match-length. For simple Pascal programs, a minimum match length between 3 to 5 is recommended. Meanwhile, for simple LISP programs, a value around 10 is recommended. The user should adjust these recommended values based on the increasing difficulties as the course proceeds.

Deimos can perform detection on one hundred LISP source code in 12 minutes. For one hundred shorter source code as Pascal program (28 lines in average), detection process is 2 minutes. The source code test samples were taken from real assignment in our class.

VI. CONCLUSION

Although the detection can be done easily, the result of automatic plagiarism detection shouldn't be used as an automated mark or as the final decision regarding whether a student has done plagiarism or not. The result should be used as an input or suggestion to help instructors doing manual investigation conscientiously. By using this tool, assessment becomes more effective and efficient.

Plagiarism detection is more useful for large programs. For small programs (short source code), the result might not be accurate. A group of short source code usually has accidental similarity between each of them since the answer is straightforward.

We developed this tool based on the real condition of programming class, and the testing also conducted using batches of source code from the assignments of the corresponding programming class. Due to the high frequency of programming assigments, the detection needs to be done easily in a short time. Moreover, source code written by beginners is simple and straightforward. Different from the previous existing plagiarism detectors, **Deimos** was specially designed to be used for this kind of basic programming courses.

Plagiarism detector **Deimos** is developed to be a part of a larger learning system, so many of its sub-functionalities can be reused or extended for other assessment functions. In the future, we plan to extend this tool by using different detections mechanism that can be used to evaluate the quality of students' source code.

Moreover, from the negative image of plagiarism and the need of detecting it, we can also get the reverse idea that our tools can be useful for learning programming. We use this "similarities" (and not plagiarism) mechanism detection as a tool for detecting students' consistency to a template (program skeleton) or standard solution. This view is more useful for beginner programming class. By having **Deimos** comparing students' source code to a given skeleton program that acts as the answer key, we can do white box assessment on students' submission.

By using automatic plagiarism detection, we hope that the rate of plagiarism will decrease and our students will put more effort in doing the assignments by themselves, so that it will increase their level of understanding about programming and also their academic achievement.

REFERENCES

[1] S. Hannabuss, "Contested texts: issues of plagiarism," Library Management, MCB University Press, vol. 22(6-7), pp. 311-318, 2001

[2] M. Joy and M. Luck, "Plagiarism in Programming Assignments," IEEE Transactions of Education, vol. 42(2), pp. 129-133, 1999

[3] S. Mann and Z. Frew, "Similarity and originality in code: plagiarism and normal variation in student assignments," Proceedings of the 8th Australian conference on Computing education, vol. 52, pp. 143-150, 2006

[4] N. Wagner, "Plagiarism by Student Programmers," 2000, http://www.cs.utsa.edu/~wagner/pubs/plagiarism0.html

[5] A. Christian and S. M. M. Tahaghoghi, "Plagiarism detection across programming languages," Proceedings of the 29th Australasian Computer Science Conference, vol. 48, pp. 277-286, 2006

[6] G. Whale, "Plague : plagiarism detection using program structure," Dept. of Computer Science Technical Report 8805, University of NSW, Kensington, Australia, 1988

[7] M. J. Wise, "Detection of Similarities in Student Programs: YAP'ing may be Preferable to Plague'ing," ACM SIGSCE Bulletin(proc. of 23rd SIGCSE Technical Symp.), vol. 24(1), pp. 268-271, 1992

[8] P. Lutz, M. Guido, and M. Phlippsen, "JPlag: Finding plagiarisms among a set of programs," Fakultät für Informatik Technical Report 2000-1, Universität Kalrsruhe, Karlsruhe, Germany, 2000

[9] M. J. Wise, "YAP3: Improved Detection of Similarities in Computer Programs and Other Texts," SIGCSE'96, p. 130-134, 1996

[10] M. J. Wise, "Neweyes: A System for Comparing Biological Sequences Using the Running Karp-Rabin Greedy String Tiling Algorithm," Department of Computer Science, University of Sydney, Australia, Technical Report 463, 1993

[11] M. J. Wise, "String Similarity via Greedy String Tiling and Running Karp-Rabin Matching," Department of Computer Science, University of Sydney, Australia, 1993

[12] R. Karp and M. Rabin, "Efficient Randomized Pattern-Matching Algorithms," IBM Journal of Research and Development, vol. 31(2), pp. 249-260, 1987

[13] J. A. Faidhi and S. K. Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment,"Computing in Education, vol. 11, pp. 11-19, 1987