Generating Pylogenetic Tree of Homogeneous Source Code in a Plagiarism Detection System

Jeong-Hoon Ji, Su-Hyun Park, Gyun Woo*, and Hwan-Gue Cho

Abstract: Program plagiarism is widespread due to intelligent software and the global Internet environment. Consequently the detection of plagiarized source code and software is becoming important especially in academic field. Though numerous studies have been reported for detecting plagiarized pairs of codes, we cannot find any profound work on understanding the underlying mechanisms of plagiarism. In this paper, we study the evolutionary process of source codes regarding that the plagiarism procedure can be considered as evolutionary steps of source codes. The final goal of our paper is to reconstruct a tree depicting the evolution process in the source code. To this end, we extend the well-known bioinformatics approach, a local alignment approach, to detect a region of similar code with an adaptive scoring matrix. The asymmetric code similarity based on the local alignment can be considered as one of the main contribution of this paper. The phylogenetic tree or evolution tree of source codes can be reconstructed using this asymmetric measure. To show the effectiveness and efficiency of the phylogeny construction algorithm, we conducted experiments with more than 100 real source codes which were obtained from East-Asia ICPC (International Collegiate Programming Contest). Our experiments showed that the proposed algorithm is quite successful in reconstructing the evolutionary direction, which enables us to identify plagiarized codes more accurately and reliably. Also, the phylogeny construction algorithm is successfully implemented on top of the plagiarism detection system of an automatic program evaluation system.

Keywords: Asymmetric local alignment, evolution process, phylogeny of source codes, plagiarism detection, source code similarity.

1. INTRODUCTION

Recently, bio-inspired computing has become a hot issue in general computer science and engineering. Especially concepts from the artificial life model give interesting insights and a new methodology for computer scientists [1]. Evolution, the most fundamental concept of biology, also represents a useful modeling framework for many computer science applications, especially in the software evolutionary process.

Constructing phylogenies of *in-silico* creatures is a very interesting means to understand the basic evolutionary process of these artificial creatures. For example, it is very important to reconstruct the phylogeny of computer viruses, since this enables anti-virus scanners to function efficiently. According to recent work, the number of new computer viruses released per week is more than 30 in the world [2]. Computer virus classification enables us to manage numerous computer viruses in a sizable library.

Tracking the evolution of computer viruses should inevitably be performed on a binary level. A group of homologous viruses is generally detected in a single query for a *conserved* string (in executable code) among several computer viruses. Kalim *et al*. proposed a new method for constructing the malware phylogeny using permutations of code [7].

Another direction for tracking the evolution of software is on a macroscopic level. Kemerer and Slaughter discussed how to trace and measure improvements of software structure [8]. A simple definition of software evolution is that it is the dynamic behavior of programming systems as they are maintained and enhanced over their life-times. Fig. 1 shows the phylogenetic graph for a set of UNIX system software to make an understanding the hierarchical structure of UNIX easily.

In this paper, we focus on the microscopic process of evolution at the source level rather than macro-

Manuscript received November 5, 2007; revised September 10, 2008; accepted November 3, 2008. Recommended by Guest Editor Phill Kyu Rhee. This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD, Basic Research Promotion Fund) (KRF-2007-314-D00232).

Jeong-Hoon Ji, Su-Hyun Park, Gyun Woo, and Hwan-Gue Cho are with the Graduate School of Computer Engineering, Pusan National University, Jangjeon-gu, Geumjeong-dong, Pusan 609-735, Korea (e-mails: {jhji, shpark07, woogyun, hgcho}@pusan.ac.kr).

* Corresponding author.

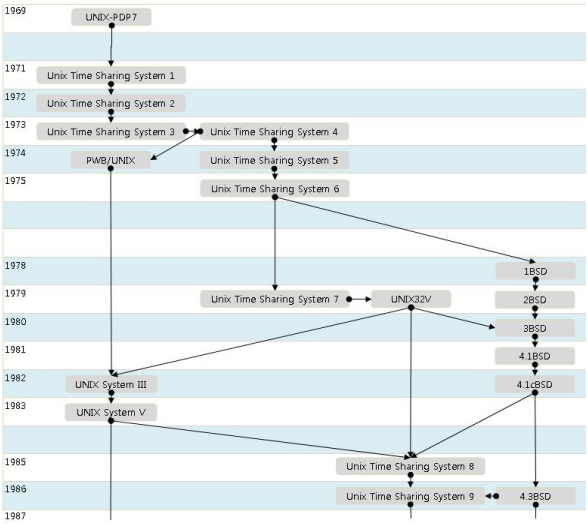


Fig. 1. A typical evolution graph of UNIX systems. Vertical direction denotes the time lines from 1969 to 1987 (years).

scopic studies by Kemerer and Slaughter [8]. The basic idea of the authors is that plagiarism of source code can be regarded as an evolutionary process for an artificial life form. In a similar manner to the means by which parts of a virus DNA are deleted, mutated, or inserted, parts of a source code (including reserved words in a high-level programming language, i.e., `for`, `if`, and `while`) are inserted, deleted, or modified while developing or improving the software.

2. MAIN PROBLEMS

In this section, we review previous work on plagiarism detection of source codes and summarize the remaining problems. The plagiarism detection methods can be categorized into two groups: non-structural methods and structural methods. The non-structural methods do not consider the structural characteristics of programs. For instance, comparing the number of words in two programs is a kind of non-structural methods. A well-known non-structural method is called attribute counting [11-14].

In the attribute counting methodology, the vector of program properties, viz., the attribute vector, is defined for each program. Then the attribute vectors of programs are compared. This method can be effective for categorizing programs but not so effective for detecting plagiarism since it cannot locate the exact plagiarized region.

Structural methods compare the structures of programs. Typical examples of structures of programs are the sequences of tokens [3,10,19], the syntax tree kernels [15,16], and call graphs of programs. Structural methods are relatively difficult to implement, but more effective at detecting plagiarism of sources.

The aforementioned previous work only addresses the similarity of two given programs, but these methods have difficulties with inferring the evolutionary direction. In some cases, the evolutionary direction can be more important than the degree of code similarity. For instance, let us assume that a source code is stolen and reproduced. Then any digital forensic tool should be able to report 'who copied what.' So, in order to infer the evolutionary direction, an asymmetric similarity metric is required. Let us summarize the two main problems unraveled in this paper.

Problem 1: Computing Asymmetric Similarity Distance between Two Different Source Codes
Input: Programs *A* and *B* are given. We know that program *B* has evolved (or been plagiarized) from program *A* at the source code level. To infer the evolutionary direction, the similarity between *A* and *B*, based on *B*, should be computed differently from the similarity between *B* and *A*, based on *A*.
Output: Two asymmetric similarity values indicating the degree to which *A* is similar to *B*, and vice versa.

Problem 2: Evolutionary Phylogeny Construction for Homogenous Codes
Input: A set of homogeneous codes which are derived from an original code is given. One typical homogenous code set includes all intermediate codes submitted in any programming contest as a solution code for a specific problem.
Output: A phylogenetic forest, to show the derivation history of input source codes.

One important aspect of the two problems explained above is how to measure the reliability of the solution. For a pair of given arbitrary programs, the solution to Problem 1 always returns two asymmetric similarity values. So we need to infer the evolutionary direction by comparing the similarity distances of one to another and vice versa. However, if two input programs are not related to each other in the development procedure of them, then the evolutionary direction could be inaccurate. So, that is the reason the input sources should be homogenous codes. Here, we explain the means to compute the asymmetric similarity metric for two different codes.

3. ASYMMETRIC SIMILARITY MEASURE FOR CODE PLAGIARISM

It is generally accepted that all kinds of evolu-

tionary processes imply a direction for adaptation. Most previous work about plagiarism detection only focused on the means to compute the similarity between programs A and B in terms of global aspects. So, they try to answer these kinds of questions: "Is program A is similar to B? If so, how similar are they?" This simple symmetric approach does not give any clues on the question of whether A is derived from B or vice versa.

But, in order to trace the evolutionary (plagiarism) history of a source code, the similarity measure must provide not only the degree of similarity, but also its direction. Here, this section explains the means to give a similarity measure with a direction, using the plain local alignment algorithm, which is a basic tool such as BLAST [20] for biological sequence (i.e., DNA, RNA, and protein) analysis.

Since our method involves exploiting local alignment for two linear keyword sequences obtained from target programs, we must construct a representative linear sequence of keywords [3]. Let L_a and L_b be the linear sequences of keywords obtained from P_a and P_b, respectively. Then we can regard the linear sequence L_x of program keywords as the "DNA" sequence of program P_x based on the analogy of the DNA sequence of a biological organism. There are several procedures for extracting the DNA of a program, but we do not consider this issue further in this paper.

As mentioned previously, local alignment has been widely applied to finding a similar region in two linear strings [9]. Local alignment is also our basic framework of plagiarism detection. Every local alignment method has its own scoring matrix, which generally determines the means to compute the similarity thus locating the exact similar aligned region.

The general local alignment algorithm uses a fixed scoring matrix (e.g., +1 denotes a match, –1 denotes a mismatch, and –2 denotes a gap insertion or deletion) which is a diagonally symmetric matrix [17]. The scoring matrix we used has two different features. First, our alignment scoring matrix is not symmetric, which implies that the matching score between two characters a and b is dependent on its direction. So, $MatchScore(a,b) \neq MatchScore(b,a)$. This enables us to evaluate asymmetric alignment scores.

Second, our scoring matrix W_D is adaptively constructed from the frequency of keywords in a program group D. Let D be a group of the functional-equivalent programs such as a set of programs submitted for an assignment task. Let the normalized frequency of a keyword K_i be f_i in D. For scoring matrix, we set $-\alpha \cdot (\log_2 f_i \cdot f_i)$ for matched keywords K_i, $+\beta \cdot (\log_2 f_i \cdot f_i)$ for mismatched keywords, $\gamma \cdot \log_2 f_i$ for gap insertion, and δ for gap deletion, where α, β, γ, and δ are the control parameters for optimizing the detection performance. In summary, the scoring matrix can be defined as:

$$W_D[x,y] = \begin{cases} -\alpha \cdot \log_2 (f_x \cdot f_y) & \text{if } x = y \\ \beta \cdot \log_2 (f_x \cdot f_y) & \text{if } x \neq y \\ \gamma \cdot \log_2 f_x & \text{if } y \text{ is a gap symbol} \\ \delta \cdot \log_2 f_y & \text{if } x \text{ is a gap symbol.} \end{cases}$$

This adaptive scoring matrix implies that the insertion or the deletion of a frequent keyword does not have less weight than those of infrequent keywords, since we believe that infrequent terms or keywords in a programming language are more crucial to the functional aspects of programs [4].

Our experiment validated that our adaptive scoring matrix is more effective than a simple plagiarism technique such as inserting meaningless/dummy keywords. This adaptive scoring matrix ensures that the matching score (and mismatching penalty) of important keywords (keywords with the lowest frequency) is higher than that of frequent ones. As shown in Table 1, inserting an uncommon `switch` results in a penalty that is 4 times higher than for inserting abundant keywords such as '=' and '`{`'.

Table 1 clearly shows that the adaptive scoring matrix is entirely dependent on the characteristics of each program group. These four control parameters, α, β, γ, and δ, are determined empirically from several experiments. We expect that $\gamma < \delta$, since it is more difficult to delete a keyword than to insert a keyword, without understanding the logical structure of a program to maintain the same function. Computing the optimal values for these four parameters is also an interesting problem. The tuning result is described in Section 5. Here, we propose an asymmetric similarity score, *AsymScore*( ) to compute the similarity of programs.

Definition 1: $AsymScore(P_a,P_b)$ is the score of a maximal local alignment using W_D (the asymmetric scoring matrix) between two keyword strings of L_a and L_b that are obtained from the programs P_a and P_b.

The evolutionary distance is defined in terms of the matching score *AsymScore*( ). Usually, the normalized

Table 1. High/low frequency keywords in a program group submitted in ICPC 2006. The total number of keywords is 9993.

Highest	Frequency	Lowest	Frequency
"="	14.00%	"`switch`"	0.01%
"`{`"	11.83%	"`-=`"	0.02%
"`}`"	11.83%	"`void`"	0.02%
"++"	6.76%	"`goto`"	0.03%
"`if`"	6.44%	Bit OR "`|`"	0.03%

similarity of two programs P_a and P_b is defined by the ratio of the matching score of the aligned region to the maximum possible matching score. So we newly define $Asym(P_a,P_b)$, which is a normalized asymmetric measure for the similarity of the program P_a and P_b based on P_b.

$$Asym(P_a,P_b) = \frac{2 \cdot AsymScore(P_a,P_b)}{AsymScore(P_a,P_a) + AsymScore(P_b,P_b)}$$

It is clear that $Asym(P_a,P_b) \sim [0,1]$. Here, using the $Asym(\ )$ function, we can define the evolutionary distance as follows:

$$EvolDist(P_a,P_b) = 1 - Asym(P_a,P_b).$$

$EvolDist(P_a,P_b)$ can be considered as the estimated amount of manual work needed to transform program P_a into P_b by keyword insertion, deletion, and exchange, while maintaining the same functionality. We adjust four weighting constants α, β, γ, and δ for match, mismatch, insertion, and deletion, respectively, in order to maximize the success rate of the phylogeny inference.

Our local alignment has two features that are different from the previous standard symmetric local alignment algorithm; (1) it uses an adaptive scoring matrix and (2) it attributes different weights for gap insertions and deletions in local alignment, thus, $EvolDist(P_a,P_b) \neq EvolDist(P_b,P_a)$ for two different programs P_a and P_b. So, it is quite natural to assume that P_b is derived from P_a if $EvolDist(P_a,P_b) < EvolDist(P_b,P_a)$. This is the most important contribution of our paper to establishing the evolution of programs. So, we address the following claim.

Claim 1: If $EvolDist(P_a,P_b) < EvolDist(P_b,P_a)$ then we assert that it is more likely that P_b is plagiarized from P_a than P_a is plagiarized from P_b.

The basic idea of our evolutionary analysis for source codes is that we try to construct a most likely evolutionary tree, using all pair-wise $EvolDist(P_i,P_j)$ values. The correctness of this claim for evolutionary direction is tested and analyzed in Section 5.

4. INFERRING ALGORITHM FOR PHYLOGENETIC TREE

In order to reconstruct the phylogeny of source codes, two important data elements should be determined: (1) the similarity distance between two source codes and (2) the direction of the influence from one to the other, given two similar source codes. The similarity distance is described in Section 3, using the function $EvolDist(\ )$.

The evolutionary direction can be determined directly and easily from the evolutionary distances. Given two programs P_a and P_b, we compute two asymmetric evolutionary distances $EvolDist(P_a,P_b)$ and $EvolDist(P_b,P_a)$. If $EvolDist(P_a,P_b) < EvolDist(P_b,P_a)$, then it is reasonable to assert that P_b has evolved from P_a than the converse since $EvolDist(P_a,P_b)$ indicates how easily P_b can be derived from P_a. We take the shorter of two evolutionary distances between the two source codes.

The evolutionary direction is a natural result of this selection, for example, if $EvolDist(P_a,P_b)$ is used, then P_b has evolved from P_a, in short $P_a \mapsto P_b$. Note that the evolutionary distance can be zero, if two programs are identical and these cases can be easily excluded.

Once the evolutionary directions and distances are computed, the evolutionary graph can be constructed. The phylogeny should be a subgraph of this evolutionary graph. After the evolutionary directions of all pairs of programs have been determined, we have a directed graph for which the undergraph is complete. We can define the phylogenetic tree as the minimum spanning tree of the undergraph without removing the directions of edges [5]. Once the phylogenetic tree has been obtained, it can be further improved by removing multiple entries. Since it is unnatural to assume that a program is derived from multiple programs, the multiple entries for a node are

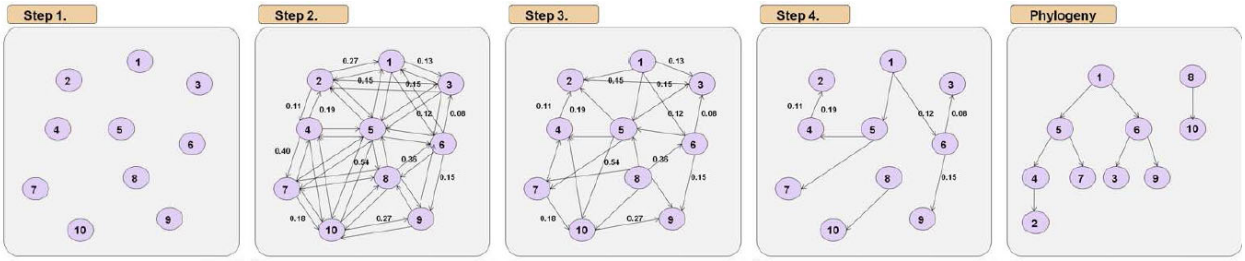
Algorithm 1: Phylogenetic Forest Construction Algorithm

procedure PHYLOFOREST(V)
- $F \leftarrow FrequencyVector(V)$;
- $M \leftarrow SimilarityMatrix(F)$;
- $E \leftarrow \left\{ \overrightarrow{(v_i, v_j)} \middle| v_i, v_j \in V, i \neq j, D(v_i,v_j) < D(v_j,v_i) \right\}$
- $T \leftarrow MinSpanningTree(Undergraph(G(V,E)))$;
- $\hat{T} \leftarrow T$
- **while** $\hat{T}$ contains a multiple-entry vertex v **do**
  - $\{e_i\} \leftarrow$ incoming edges of v;
  - $e^* \leftarrow$ maximal weight edge in $\{e_i\}$;
  - $\hat{T} \leftarrow \hat{T} - e^*$;
- **end while**
- **return** $\hat{T}$

end procedure

function $D(v_1,v_2)$
- $MaxS \leftarrow AsymScore(v_1,v_1) + AsymScore(v_2,v_2)$;
- **return** $1 - 2 \cdot AsymScore(v_1,v_2)/MaxS$;

end function



(a) Step 1. (b) Step 2. (c) Step 3. (d) Step 4. (e) Step 5.

Fig. 2. Step 1 shows the 10 given programs. Step 2 involves computing the bi-directional evolutionary distances between node *a* and *b*, which is represented by double edges. Step 3 involves selecting a more plausible evolutionary edge. Step 4 involves constructing a minimal spanning tree in the under-graph of the directed graph given in (c). Step 5 involves applying our Spanning Forest Construction Algorithm to the minimal spanning tree obtained in (d). In Step 5, this forest is the final output of this work.

removed. As a result, the phylogenetic *forest* is obtained from the program evolutionary graph [6]. Algorithm 1 shows the steps of phylogenetic forest construction.

Algorithm 1 reconstructs the phylogenetic forest $\hat{T}$ inferred from the given set of programs $V = \{v_1, v_2, \ldots, v_n\}$. To summarize, the phylogenetic tree T is the minimum spanning tree of the undergraph $G(V,E)$ where E is the set of edges for which the evolutionary distances are defined using M; and the phylogenetic forest $\hat{T}$ is an improved phylogeny, obtained by removing multiple entries from T. Fig. 2 illustrates the entire step by step procedure in our phylogeny construction algorithm (Algorithm 1).

5. EXPERIMENTS

In order to evaluate the correctness of our phylogeny inferring algorithm, we tested our algorithm, using a set of artificially plagiarized programs. We collected 20 sets of programs. Each set contains one root program (the original code) and four other programs derived from it by 20 graduate students manually. In some groups, derived programs were constructed via sequential modifications such that $P_0 \mapsto P_1 \mapsto P_2 \mapsto P_3 \mapsto P_4$. $A \mapsto B$ denotes that program B was derived via modification of program A. Another test groups contain five programs, where $P_0 \mapsto \{P_1, P_2\}, P_2 \mapsto \{P_3, P_4\}$. These derivation procedures can be completely described in a tree model, viz., phylogenetic tree. So we can construct the "true phylogenetic tree" for each group of artificially plagiarized codes. Here, we are ready to test the phylogenetic tree construction algorithm.

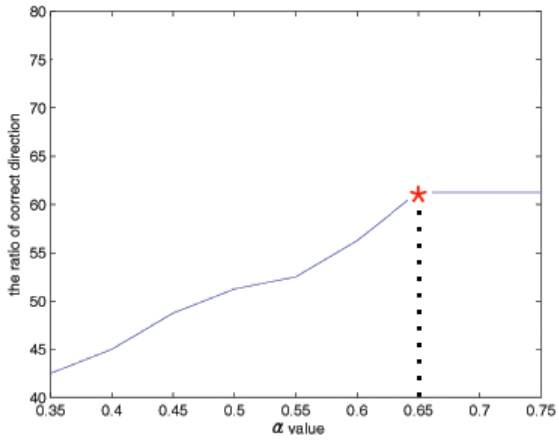
The simplest criteria for evaluating the correctness of a phylogenetic tree is the count of the number of inversion tree edges. There is one correct (real) phylogenetic tree with five nodes and four directed edges, for each program group. Therefore, the total number of evolutionary edges in the 20 test groups should be 80. The inversion edge is defined as the directed edge with the wrong time direction. For each test program, the subscript t of P_t means the time-index for program generation. This implies that the appearance time of P_0 precedes that of P_1, P_2, P_3, and P_4; and P_2 also precedes P_3 and P_4 etc. So, for a tree generated by our algorithm, if we obtaining a directed edge $P_i \mapsto P_j$, where $i < j$, then we consider this edge to be correct with respect to the evolutionary procedure. Otherwise, if we obtaing an edge $P_i \mapsto P_j$ where $i > j$, then we call this edge an inversion edge or simply an "inversion." Thus, the objective function for inferring the phylogenetic tree can be reduced to minimizing the number of inversion edges.

5.1. Control parameter tuning for phylogenetic tree optimization

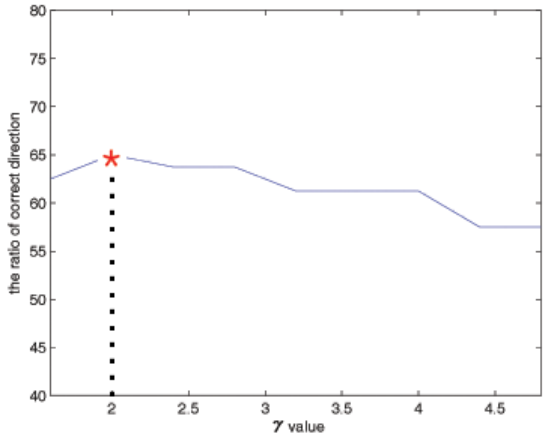
The performance of local alignment is completely dependent on a few control parameters. So, prior to applying the phylogenetic tree generation algorithm, we must find the nearly optimized control parameters, to compute the asymmetric similarity. Since this problem is a kind of multi-variable, non-linear optimization, there is no sound optimization procedure for local alignment.

In this work, we implemented a very simple method, sequential parameter optimization. We only consider three control parameters: α for the matching gain, δ for the gap insertion penalty, and γ for the gap deletion penalty (refer to Section 2). In this procedure, we fixed the penalty value for a mismatch $\beta = 0.35$, which was obtained empirically by experiments in the authors' previous work [3,4].

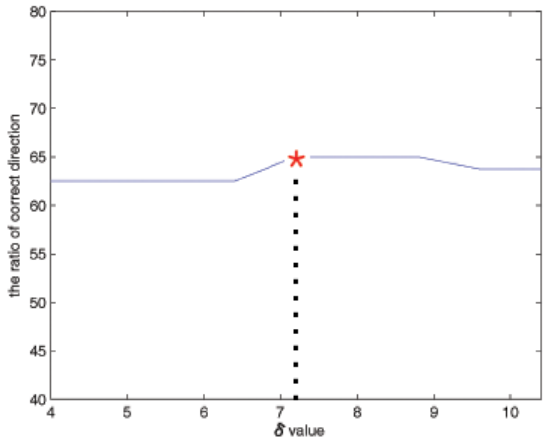
The order of tuning parameters was α, δ, and γ. For each parameter, we tried to minimize the number of correctly inferred evolutionary edges by increasing the parameter value by a small unit amount. The



(a) Optimizing parameter α.



(b) Optimizing parameter γ.



(c) Optimizing parameter δ.

Fig. 3. Parameter tuned result α = 0.65, γ = 2.0, and δ = 7.2.

parameter tuning result is shown in Fig. 3. Subsequent to tuning α, we tuned γ. And, with the previously determined values for α and γ, we tried to optimize δ. Via this sequence of optimization we finally obtained $\alpha = 0.65$, $\delta = 2.0$, and $\gamma = 7.2$.

5.2. Constructing program phylogeny

Table 2 shows the basic statistics for the four test program groups. Four root (origin) source codes were selected from the source code submitted as correct solutions in ICPC 2006. In order to construct homogeneous groups, the origin source P_1 is transformed into two different programs P_2, P_3, and one of P_2 and P_3 is selected and transformed again into other two programs P_4 and P_5. Therefore we could construct four evolution edges by this derivation, which could be a true edge set.

Table 2. Test data for testing the correctness of inferred phylogeny by the proposed algorithm.

Groups	Codes	Trees	Lines of Source Code			
			Min	Max	μ	σ
P01	25	5	91	234	120.40	27.95
P02	25	5	39	110	86.20	17.81
P03	25	5	65	213	103.44	29.08
P04	25	5	72	227	111.20	30.36

By adopting the tuned parameters, our asymmetric measure could correctly infer about 67% of evolutionary directions, among 80 evolutionary directions. The estimation ratio differs according to the program group. For program group P01, P02, P03, and P04, 80%, 55%, 70%, and 55% of directions were correctly estimated, respectively.

Fig. 4 shows some examples of phylogenetic trees of homogeneous test codes. In the phylogenetic tree, each node denotes the corresponding source code and the edge direction implies the evolutionary direction computed. The node labels of the tree in Fig. 4(a) denote the time index, in terms of the evolutionary procedure. The prefix of the node label denotes the ancestry of the program. For example, the program labeled **11** is evolved from the program labeled **1** and the program labeled **112** is evolved from the program labeled **11**. In Fig. 4(a), it is clear that the evolutionary direction is completely correct; but in Figs. 4(b) and (c), there are some incorrect directions, which are denoted by dashed edges.

The edge weight shown in Fig. 4 implies that the normalized evolutionary distance metric, and the distance ranges from zero (completely identical) to one (completely different).

5.3. Application: Detecting code plagiarism

We used the phylogeny construction algorithm to perform, plagiarism detection among source codes submitted in an International Collegiate Programming Contest (ICPC). Since the ICPC final competition took place in a highly secure environment, we did not find any plagiarized codes. Rather than just obtaining the final codes submitted, we obtained numerous homogenous codes (a set of tried codes that were submitted as a solution program to a designated contest problem). These homogenous test codes were successfully clustered and their sequence of

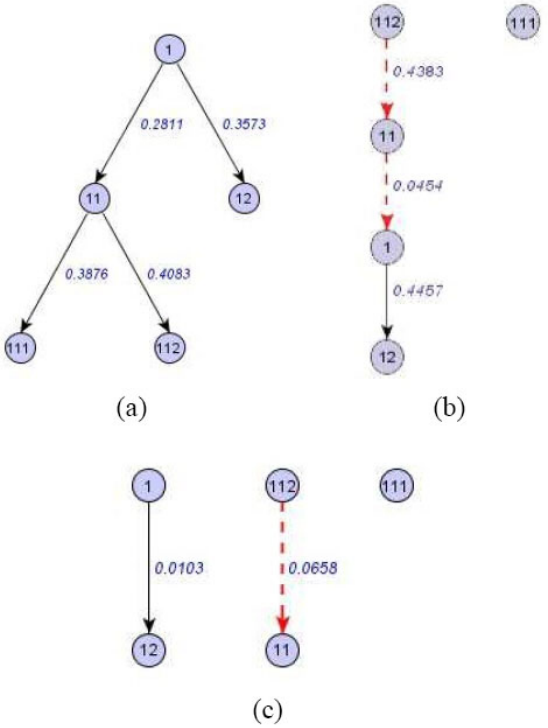


Fig. 4. Reconstructed evolutionary trees with plagiarized homogeneous codes: a correct phylogeny (a) and two incorrect phylogenies (b) and (c). The tree node denotes a source code, and an edge denotes the direction of evolution. Edge labels denote the evolutionary distances.

submission was successfully inferred by our algorithm.

Since the preliminary round of every ICPC take place in a distributed environment, due to the huge number of participants, some plagiarism is likely to occur. So, every participant can access the contest problem and submit the solution programs from their home or university computer lab, or even from a commercial Internet cafe. So, since the preliminary round of ICPC is unmonitored by a supervisor, we expect a few cases of cheating (plagiarism) to occur during ICPC.

Table 3 shows the entire set of test data (source code) obtained from ICPC. In the preliminary Internet competition of ICPC 2005: Problem-E, our asymmetric code similarity measure reported that there were eight program pairs with very high similarity scores. We manually investigated each pair of similar programs, then, we finally exposed real-world code plagiarism, where the participants that cheated confessed that one student had sent the correct solution using an instant Internet messenger.

Surprisingly, this year, in the ICPC 2007 final round, our algorithm detected one instance of Internet-based cheating. One team had hacked the competition system (PC^2, which was developed and released by IBM) using packet sniffing during the competition.

Table 3. Experimental codes for detecting program plagiarism by our algorithm.

Group	Programs	Pairs	Avg. Lines
ICPC2004-B	48	2256	89.18
ICPC2004-C	22	462	55.17
ICPC2004-E	35	1190	44.44
ICPC2005-A	153	23256	65.30
ICPC2005-B	109	11772	67.49
ICPC2005-E	38	1406	44.14
ICPC2005-G	44	1892	47.60
ICPC2006-A	180	32220	43.77
ICPC2006-B	175	30450	54.29
ICPC2006-C	157	24492	58.98

The dishonest team modified the stolen code and submitted it to the judge. Our tool was quite successful in isolating such a group of dishonest students.

6. APPLICATION: AUTOMATED EVALUATING SYSTEM FOR PROGRAMMING ASSIGNMENT WITH ANTI-PLAGIARISM FEATURES

We developed a web-based automated Evaluating System for Programming Assignments (ESPA), which is actually used in introductory C/C++ and Java programming language courses in our department. This system executes the programs submitted by the students and evaluates them automatically, by comparing the results of the students' program with the correct solution given by the professor. The ESPA system also checks if there are suspected plagiarized program pairs after the submission dead-line is over. Fig. 5 shows an example of the submission tracking

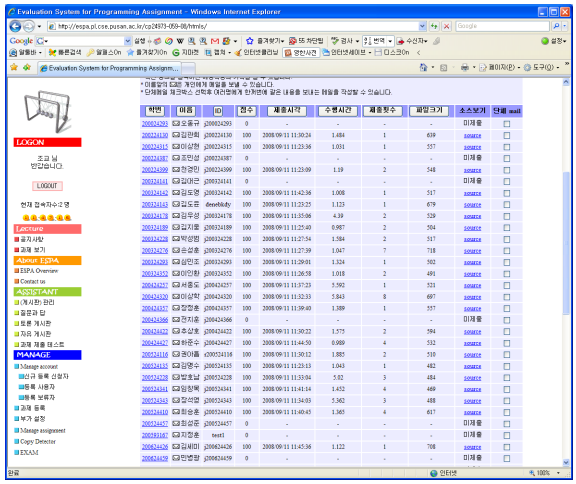


Fig. 5. A screenshot of the Evaluating System for Programming Assignments. The submission time, the number of submission, and the score of each assignment are displayed.

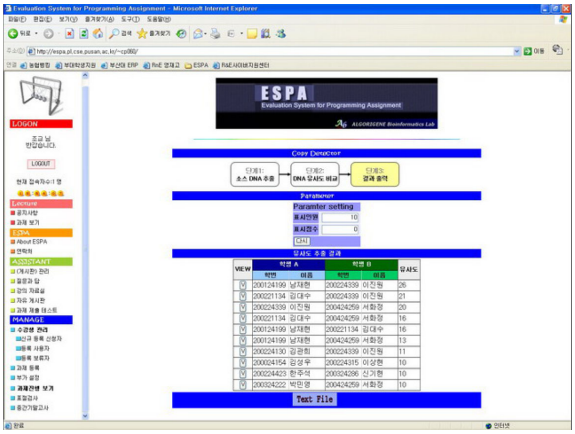


Fig. 6. The screenshot for plagiarism test procedure in ESPA.

page of ESPA and Fig. 6 shows an example of the plagiarism detection page of ESPA.

Prior to using this plagiarism investigation system, we determined that around 5% of all enrolled students (60–80 for a class) seemed to have tried to cheat in their programming assignment by plagiarizing another person's code. But, after notifying that ESPA would check for code plagiarism, the number of cheating students decreased dramatically, to nearly zero for a semester. Especially, we found that merely showing an evolutionary tree for the cheating program group to students was quite successful at determining dishonest behavior. We finally established that this kind of automated grading system with anti-plagiarism features is quite effective and efficient in introductory programming education.

7. CONCLUSION

In this paper, we proposed a phylogenetic forest generation algorithm based on bio-inspired local alignments. The local alignment algorithm is modified to generate an asymmetric similarity score for two source codes. According to the experimental results, the proposed algorithm is quite good at inferring evolutionary directions of programs. We summarize the main contributions of our work as follows:

- We proposed an evolutionary distance measure with a direction to infer the degree and direction of evolution among source codes.
- Our experiment showed that this measure archived more than 67% accuracy at reconstructing the evolutionary direction among homogeneous codes.
- This result implies that our evolutionary measure can be successfully applied to detection of plagiarized pairs of programs among heterogeneous codes.
- So, our measure can be used to investigate cheating by dishonest students in a programming assignment. The authors have empirically confirmed that plagiarism detection and evaluation analysis based on this work are very efficient and effective means to deter dishonest students.

Bio-inspired evolutionary analysis can be widely exploited in numerous types of software analysis, such as code maintenance, source clustering, plagiarism detecting. Currently, we are developing an authorship analysis scheme based on the proposed phylogeny generation algorithm. We hope this type of evolutionary analysis for source codes will become an integral part of future digital forensic work.

REFERENCES

[1] N. Forbes, *Imitation of Life: How Biology is Inspiring Computing*, MIT Press, 2004.

[2] L. A. Goldberg, P. W. Goldberg, C. A. Phillips, and G. B. Sorkin, "Constructing computer virus phylogenies," *J. of Algorithms*, vol. 26, no. 1, pp. 188-208, January 1998.

[3] J.-H. Ji, G. Woo, and H.-G. Cho, "A source code linearization technique for detecting plagiarized programs," *ACM SIGCSE Bulletin*, vol. 39, no. 3, pp. 73-77, June 2007.

[4] J.-H. Ji, G. Woo, S.-H. Park, and H.-G. Cho, "An intelligent system for detecting source code plagiarism using a probabilistic graph model," *Proc. of the 5th International Conference on Machine Learning and Data Mining in Pattern Recognition*, MLDM Posters 2007, pp. 55-69, July. 2007.

[5] J.-H. Ji, S.-H. Park, G. Woo, and H.-G. Cho, "Evolution analysis of homogenous source code and its application to plagiarism detection," *Proc. of the FBIT2007*, pp. 813-818, October 2007.

[6] J.-H. Ji, G. Woo, S.-H. Park, and H.-G. Cho, "Understanding evolution process of program source for investigating software authorship and plagiarism," *Proc. of the 2nd International Conference on Digital Information Management*, pp. 98-103, October 2007.

[7] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *J. in Computer Virology*, vol. 1, no. 1, pp. 13-23, 2005.

[8] C. F. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *IEEE Trans. on Software Engineering*, vol. 25, no. 4, pp. 493-509, 1999.

[9] S. Meyer zu Eissen and B. Stein, "Intrinsic plagiarism detection," *Proc. of ECIR 2006, Lecture Notes in Computer Science*, vol. 3936, pp. 565-569, 2006.

[10] L. Prechelt, G. Malpohl, and M. Philippsen,

"Finding plagiarisms among a set of programs with JPlag," *J. of Universal Computer Science*, vol. 8, no. 11, pp. 1016-1038, 2002.

[11] J. H. Johnson, "Identifying redundancy in source code using fingerprints," *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 171-183, IBM Press, 1993.

[12] S. Brin, J. Davis, and H. García-Molina, "Copy detection mechanisms for digital documents," *Proc. of the ACM SIGMOD Annual Conference*, pp. 398-409, 1995.

[13] K. L. Verco and M. J. Wise, "Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems," *Proc. of the 1st Australian Conference on Computer Science Education*, Sydney, Australia, pp. 130-134, July 1996.

[14] S. D. Stephens, "Using metrics to detect plagiarism (student paper)," *Proc. of the 7th Annual Consortium for Computing in Small Colleges*, pp. 191-196, Consortium for Computing Sciences in Colleges, USA, 2001.

[15] I. D. Baxter, A. Yahin, L. M. D. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," *Proc. of the International Conference on Software Maintenance*, pp. 368-377, 1998.

[16] J.-W. Son, S.-B. Park, and S.-Y. Park, "Program plagiarism detection using parse tree kernels," *Proc. of the 9th Pacific Rim International Conference on Artificial Intelligence, Lecture Notes in Computer Science*, Springer, vol. 4099, pp. 1000-1004, Aug. 2006.

[17] M. J. Wise, "YAP3: Improved detection of similarities in computer program and other texts," *Proc. of SIGCSE '96*, pp. 130-134, 1996.

[18] A. Aiken, *Moss (Measure of Software Mimilarity) Plagiarism Detection System*, Available: http://theory.stanford.edu/~aiken/moss/, 1998.

[19] D. Gitchell and N. Tran, "Sim: A utility for detecting similarity in computer programs," *Proc. Of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, pp. 266-270, ACM Press 1999.

[20] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J. Molecular Biology*, vol. 215, pp. 403-410, 1990.

Jeong-Hoon Ji received the B.S. degree in 2003 from Kyungsung University, Korea and the M.S. degree in 2005 from Kyungsung University, Korea. Since 2005 he has been a doctoral candidate student in Pusan National University, Korea. His research interests are programming language and software plagiarism detection.

Su-Hyun Park received the B.S. degree in 2007 from Pusan National University, Korea. Since 2007 she has been a master course student in Pusan National University, Republic of Korea. Her research interests are scientific visualization and bioinformatics.

Gyun Woo received the B.S., M.S., and Ph.D. degrees from Korea Advanced Institute of Science and Technology, Korea in 1987, 1991, and 2000 respectively. Since 2004 he has been a Professor in Pusan National University, Korea. His research interests are functional programming and program analysis.

Hwan-Gue Cho received the B.S. degree in 1984 from Seoul National University, Korea, the M.S. degree in 1986 from Korea Advanced Institute of Science and Technology, Korea, and the Ph.D. degree in 1990 from Korea Advanced Institute of Science and Technology, Korea. Since 1990 he has been a Professor in Pusan National University, Korea. His research interests are graphics (visualization) and bioinfor-matics (sequence alignment and bionetwork analysis).