The Computer Journal Advance Access published June 24, 2005

doi:10.1093/comjnl/bxh119

PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets

LEFTERIS MOUSSIADES[1] AND ATHENA VAKALI[2]

[1]*Division of Computing Systems, Department of Industrial Informatics, Technological Educational Institute of Kavala, GR-65404 Kavala, Greece*
[2]*Department of Informatics, Aristotle University, 54124 Thessaloniki, Greece*
Email: lmous@teikav.edu.gr, avakali@csd.auth.gr

Efficient detection of plagiarism in programming assignments of students is of a great importance to the educational procedure. This paper presents a clustering oriented approach for facing the problem of source code plagiarism. The implemented software, called PDetect, accepts as input a set of program sources and extracts subsets (the clusters of plagiarism) such that each program within a particular subset has been derived from the same original. PDetect proposes the use of an appropriate measure for evaluating plagiarism detection performance and supports the idea of combining different plagiarism detection schemes. Furthermore, a cluster analysis is performed in order to provide information beneficial to the plagiarism detection process. PDetect is designed such that it may be easily adapted over any keyword-based programming language and it is quite beneficial when compared with earlier (state-of-the-art) plagiarism detection approaches.

Received 15 June 2004; revised 31 May 2005

1. INTRODUCTION

It is common knowledge among the higher education community that a number of students are engaged in some forms of academic dishonesty [1]. Plagiarism in programming assignments is a conventional form of such dishonesty and it constitutes a major menace to the educational procedure. Therefore, detection of plagiarism in programming assignments is an essential task for each instructor to carry out. Certainly, it is a time-consuming and toilsome task to be performed manually, since the evaluation of n programs requires the contrast of $[n*(n-1)]/2$ pairs of programs. Moreover, this task's difficulty is quite extensive owing to the fact that students tend to modify the plagiarized code in order to avoid detection. Thus the automation of plagiarism detection in programming assignments has motivated the research community since mid-1970s.

1.1. Attribute-counting systems

The early systems, known as attribute-counting systems, were based on Halstead's software science metrics [2]. According to these metrics a program P may be considered as a sequence of tokens classified as either operators or operands: N_1 is the number of operator occurrences, N_2 the number of operand occurrences, n_1 the number of distinct operators and n_2 the number of distinct operands. In [3] a four-tuple of (n_1, n_2, N_1, N_2) is assigned to each program such that programs with the same four-tuples are considered suspicious. In due course other metrics were added to achieve more accurate measure of similarity. Code lines, variables declared and total control statements were added in [4]. Total numbers of variables, subprograms, input statements, conditional statements, loop statements, assignment statements and calls to subprograms are used by [5] in order to detect similarity between Fortran programs. A measure for program structure has also been proposed by [5]. This measure is based on the assignment of a character code to certain code structures and then building a string from character codes. In [6] a minimal set of 23 measures in which no measure is correlated with any other is proposed. These measures include the average number of characters per line, the average identifier length, the number of reserved words, the conditional statement percentage etc. [6] has also added more complex counts to reflect program's flow control. This is achieved by partitioning a program's flow control into intervals based on the algorithm described in [7].

1.2. Structure metric systems

Attribute-counting systems have been shown [8] to be poor in terms of plagiarism detection performance, when compared with the more contemporary systems, known as structure metric systems, which rely on structure comparison. Structure metric systems are involved in certain implementations such as MOSS (Measure of Software Similarity) [9], YAP series (1, 2, 3) [10] and

JPlag [11]. MOSS plagiarism detection supports white space insensitivity, noise suppression and position independence. This behavior is based on a string-matching algorithm [12] that:

- divides programs into k-grams,[1]
- hashes each k-gram,
- selects a subset of these hashes to be the program's fingerprint and
- compares the fingerprints.

YAP3 converts programs into token strings and compares them by using the Greedy-String-Tiling algorithm [13] to detect shuffled code segments. The Greedy-String-Tiling algorithm compares two strings, the pattern (i.e. shorter) and the text by searching in the text to find substrings of the pattern. Matches of substrings that are shorter from the minimum-match-length parameter are ignored. Maximal matches of string sequences that start at position p in pattern and position t in text become permanent and unique associations, and they are called tiles. JPlag uses the same algorithm as YAP3 optimizing its run time complexity. All these systems receive as input a set of programs and output a set of program pairs each of which is associated to a similarity value. A critical parameter for structure metric systems that is used to avoid spurious matches is the minimum-match-length. A feature of structure metric systems is that they support the visualization of program comparison by using a different font color for each particular tile.

1.3. Other approaches

A different approach is presented in [1] where XML is used to convert procedural programs into trees. The structure of these trees is then transformed into matrices (the so-called decimal frame matrices), based on specific mapping relationships. Based on these matrices, the similarity between two programs is numerically quantified. However, this approach is restricted on the procedural language model, as it supposes that every program consists of three main structure blocks: Headers, Global Variables, and Functions (each Function may be structured containing blocks of all three types). Therefore, this approach is not applicable to other than procedural programming languages.

Another relative to source code plagiarism detection work focuses on the 'one-to-many' visualization of the similarities between source files [14]. This visualization is achieved by plotting a point at coordinate (x, y) when a character sequence of k-length (a k-gram) begins at position x in the base file and it has been also found at y files in the test set. However, this visualization is based on the comparison of text files using pattern grams and it only exploits partially the specificities of the source code.

1.4. Contribution

None of the above discussed systems [1, 3, 4, 5, 6, 7, 9, 10, 11] succeed in detecting the actual clusters of plagiarism, since for a given set of sources they calculate pairwise similarities or produce 'one-to-many' file visualizations. However, detection of plagiarism clusters is very important for the following reasons.

- It is a common experience among tutors that plagiarism in programming students' assignments tends to occur in the form of clusters.
- Clustering may assist the reliability of detection in various ways. For example, two programs, which are not associated as plagiarism transcripts before clustering, may be associated after clustering as being members of the same cluster.
- Detection of plagiarism clusters has beneficial effects on both the students' progress and the instructor's strategies. More specifically, the detection of plagiarism clusters allows the tutor to provide the students with more accurate feedback, which is recognized as an effective motivational factor by most methods for designing motivational strategies [15]. It also gives the instructor the opportunity to enhance his/her educational methods, since he/she may well use the plagiarism clusters in order to set learning groups and promote cooperation between students, resulting in higher achievement and more productive learning [16].

In this context, the main contribution of this paper is summarized in:

- *proposing* an efficient attribute-counting detection approach for identifying pairs of programs that are most-likely plagiarized;
- *employing* a clustering algorithm on a set of pairs of programs which are 'suspicious' for plagiarism to identify the clusters of plagiarism;
- *introducing* the use of suitable measures for evaluating the plagiarism detection performance;
- *combining* the proposed with other earlier plagiarism detectors[2] and performing cluster analysis in order to identify the appropriate information which will facilitate the process of plagiarism detection.

The remainder of the paper is structured as follows. In Section 2, we describe the PDetect 2-phase processing that accepts as input a set of programming assignments and outputs the clusters of plagiarism. Performance measures for plagiarism detection are discussed in Section 3. The analysis of the effects of clustering in plagiarism detection performance is presented in Section 4. In Section 5 an evaluation compares PDetect performance with JPlag performance. Finally, Section 6 reports conclusions and highlights further research topics.

2. CLUSTERING PROGRAM SOURCES IN TWO PHASES

The PDetect functionality consists of the source code representation, the definition of the similarity measure and the clustering algorithm that detects the clusters of

[1] A k-gram is a contiguous substring of length k, where k is a user-defined parameter. For example, the text 'left' contains the 2-grams 'le', 'ef' and 'ft'.

[2] In this paper we use the term 'detector' to refer to software designed to detect plagiarism in source code datasets.

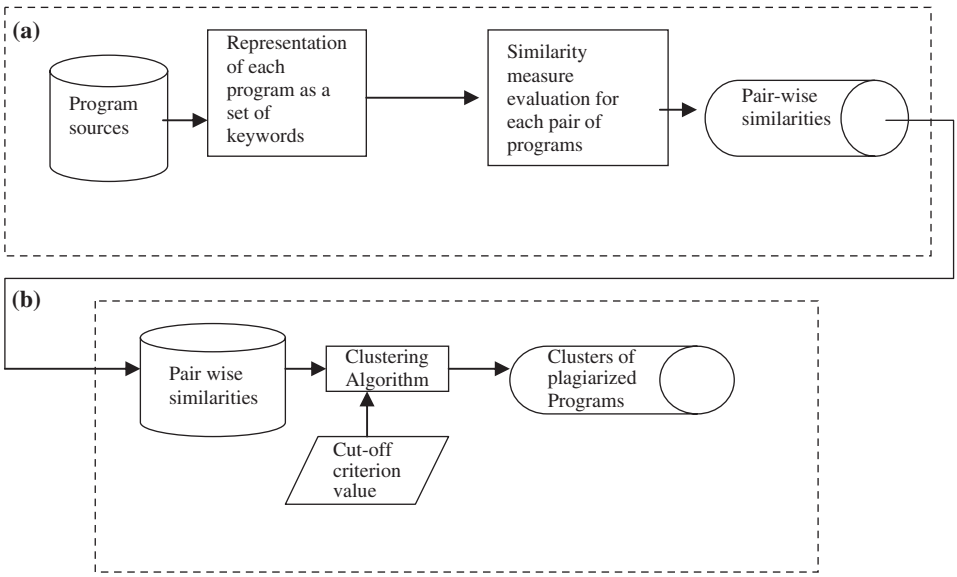

FIGURE 1. PDetect: the clustering framework. (**a**) Pairwise similarities evaluation (Phase 1). (**b**) Detection of the plagiarism clusters (Phase 2).

plagiarism. Each of these approaches is described in the following subsections.

2.1. PDetect overview

PDetect operates in two phases. Phase 1 processing extracts the pairwise similarities from a given set of programs and then in phase 2 processing the pairwise similarities are given as input and the clusters of plagiarism are detected.

- In phase 1 [Figure 1(a)], a set of program files is given as input. For each program in the set a representation based on the keywords, which are contained in the program, is produced. Then for each pair of programs a similarity measure is calculated. The results of phase 1 constitute a text file (pairwise similarities) each line of which contains the identifications of two program files and their corresponding similarity. The lines of the file are sorted by the descending order of the similarity measure. This file is provided to the user, who has to decide for each individual pair whether it is a plagiarism pair or not. This task is facilitated if the user examines program pairs starting at the one with the highest similarity value and progressing towards lower values. In this case the user may avoid examining all program pairs as it is expected that plagiarism pairs are concentrated towards higher similarity values. Thus, the user should estimate a similarity value, as a bottom threshold under which it is unlikely for plagiarism to have occurred. This threshold value is known as the *cutoff criterion value* [11]. Cutoff criterion value and pairwise similarities are given as the inputs of phase 2.
- During phase 2 [Figure 1(b)] all program pairs, which are associated with a similarity value higher than or equal to a specified cutoff criterion value, are represented as a weighted non-directed graph such that vertices represent programs and edges represent similarities between programs. The graph is then clustered using an appropriate graph-clustering algorithm.

To clarify phase 1 and phase 2 processing we focus on the following.

- The program's representation (phase 1).
- The adopted similarity measure, which is the Jaccard coefficient.
- The WMajorClust algorithm which introduces the graph-clustering idea to the plagiarized clusters detection (phase 2).

2.2. Source code representation

According to the definition of plagiarism given by Parker and Hamblen [17] a plagiarized program is a program that has been produced from another program (original) with a small number of routine modifications but with no actual understanding of the program code. This definition seems to be reasonable in the context of student programming assignments, since students who understand the program code do not usually plagiarize. Besides if a student plagiarizes a program and proceeds to structural modifications, which presuppose actual understanding of the code, then from the tutor's point of view this plagiarism may not be undesirable.

However, a major difficulty in detecting plagiarism in student programming projects arises from the students' tendency to modify the plagiarism transcript in various forms in order to avoid detection. A detailed description of all the relevant techniques, which are known as types of attacks, is presented in [11]. None of these types of attacks refers specifically to the replacement of program keywords. Thus, the core assumption of this paper is

TABLE 1. Function initGame is a plagiarism transcript of function init.

void init(**int&** start, **int&** end) **{**	**void** initGame(**int&** first, **int&** last) **{**
startGame=**false**;	**int** i=0;
for (**int** i=0; i<4; i++) players[i]=0;	**while** (i<4) **{** person[i]=0; i++; **}**
string message="Snakes and Ladders";	string gameTitle="Snakes and Ladders";
setTitle(message);	title(gameTitle);
start=1;	playOn=**false**;
end=100;	positions=0;
back=0;	first=1;
}	last=100;**}**

TABLE 2. Representation of functions init and initGame as indexed sets of keywords.

Init={void1, int1, int2, false1, for1, int3}
InitGame={void1, int1, int2, int3, while1, false1}
a: Using only language keywords. Common keywords 4. Total keywords used 6.
Init={void1, &1, &2,int1, int2, false1, for1, int3, [1,string1}
InitGame={void1, &1, &2,int1, int2, int3, while1, [1,string1, false1}
b: Using user-defined keywords. Common keywords 9. Total keywords used 11.
Init={void1, &1, &2,int1, int2, false1, loop1, int3, [1,string1}
InitGame={void1, &1, &2,int1, int2, int3, loop1, [1,string1, false1}
c: Using substitute keywords. Common keywords 10. Total keywords used 10.
Init={void1, &1, &2,&3,&4,int1, int2, false1, loop1, int3, [1,[2,string1}
InitGame={void1, &1, &2,&3,&4,int1, int2, int3, loop1, [1,[2,string1, false1}
d: Using weighted substitute keywords. Common keywords 13. Total keywords used 13.

that no type of attack causes major changes in the set of keywords that a program uses. However, a simple set of keywords ignores multiple occurrences of the same keyword. Therefore, an indexed set of keywords, which is a set of indexed keywords, is preferable. Consequently, a measure of distributional similarity between the indexed sets of keywords that represent students' programs should be a reliable indicator of plagiarism. However, standard language keywords do not always reflect precisely all the program words that may be keywords for the purpose of efficient plagiarism detection. Reciprocally, what we really need is an indexed set of substitute keywords. In Table 1, an example of two C++ plagiarized functions is presented. Function 'initGame' has been produced from function 'init' by encapsulating some common types of attack.

Any measure of similarity that could be used to compare Table 1 functions would fail to discover high correlation between them. However, on more careful observation, both of them appear to support exactly the same functionality since they both return a void value, take two integer parameters by reference, declare and initialize a string, initialize a Boolean to false, an integer to 1, another to 100 and a third to 0. Finally, they both initialize the elements of a 4-position integer array to 0 using a different loop structure. In Table 2, (a) has the representation of these functions as indexed sets of C++ keywords.

Although this representation shows high similarity (four common keywords out of six totally used), it does not take into account the references, the string declaration (since string is not a C++ keyword) and the square brackets that indicate the presence of an array. This deficiency can easily be corrected if we allow the user to define additional keywords that she/he believes are more appropriate for the plagiarism detection. In Table 3, (a) presents the subset of user-defined keywords, which are necessary for the representation of Table 1 functions.

TABLE 3. User-defined and substitute keywords.

(void,int,false,for,while,&,[,string)
a. User defined keywords
(void,int,false,(for,**loop**),(while,**loop**),&,[,string)
b. Substitute keywords
(void,int,false,(for,loop),(while,loop),(&,**2**),([,**2**),string)
c. Weighted substitute keywords

Using the new representation [Table 2 (b)] we may now calculate a higher similarity for functions 'init' and 'initGame', as their common keywords are 9 of 11 used in total. However, this representation can be improved further, taking into account a common type of attack, which is the modification of control structures. The modification of control structures is reported by [11] to be 100% successful. The most frequent modification of this type is the replacement of a for-loop by a while-loop. A replacement of this type is demonstrated in our example in Table 1. PDetect confronts this type of attack by allowing the user to declare substitute keywords as is demonstrated in Table 3 (b). In this case, the resulting sets are calculated as they are presented in Table 2 (c) and the similarity of the two sets is now 100%. Substitute keywords should also be of use in some cases of another type of attack known as the modification of data structures.

For example, the replacement of a string with an array of characters becomes detectable with the appropriate use of substitute keywords. Thus, the user of our system should declare substitute keywords taking into account these two types of attacks and the programming language used. In addition, substitute keywords may be weighted as they appear in Table 3 (c), allowing the user to determine the importance of each substitute keyword to a particular detection task. Weight for a substitute keyword should be declared higher when the use of a keyword is expected to be infrequent in a particular assignment. The default weight value is 1 for all keywords. It should be noted that although we have currently used PDetect only for C++ source files, it is easy to use it in other keyword-based programming languages by only providing the appropriate dictionary of keywords.

2.3. The similarity measure

If p_1, p_2 are two programs and $T(p)$ is the indexed set of substitute keywords of program p, a similarity measure w called the Jaccard coefficient [18] is defined by:

$$w(p_1, p_2) = \frac{|T(p_1) \cap T(p_2)|}{|T(p_1) \cup T(p_2)|},$$

where w satisfies $w(p, p) = 1$ and $w(p_1, p_2) = w(p_2, p_1)$ but does not satisfy the triangle inequality. We use the Jaccard coefficient based on its better results over an empirical comparison of seven well-known similarity measures. More precisely, the L2 norm, Kendall's τ, the confusion probability, the cosine metric, the L1 norm, Jensen–Shannon divergence, and Jaccard coefficient are compared in [19] and Jaccard coefficient appears to produce the lowest error rate.

2.4. WMajorClust: the clustering process

At phase 2, we construct a weighted non-directed graph $G = \langle V, E \rangle$ such that vertices V represent program identifiers and weighted edges E represent the similarity (evaluated by the Jaccard coefficient) between programs. In this graph we take into account program pairs only if their program members are associated with a Jaccard coefficient value higher than a considered cutoff criterion value. By using this representation, the problem of detecting plagiarism clusters has been converted to the problem of clustering the weighted graph. The proposed algorithm is called WMajorClust and it is based on the idea of using a clustering algorithm on a weighted graph, inspired from the so-called MajorClust [20] algorithm, which operates on non-weighted graphs. Therefore, WMajorClust is a graph-clustering algorithm based on the maximization of a particular connectivity measure, namely the weighted partial connectivity Λ. To formulate the problem definition, we consider G to be a graph with vertex set $V(G)$ and edge set $E(G)$. Let $C = (C_1, \ldots, C_n)$ be a decomposition [21] of G into n subgraphs (each subgraph is a cluster) where $\cup_{V(C_i \in C)} = V(G)$ and $V(C_i) \cap V(C_j)_{,i \neq j} = \emptyset$. The weighted partial connectivity of C, $\Lambda(C)$ is defined, similar to [20] as

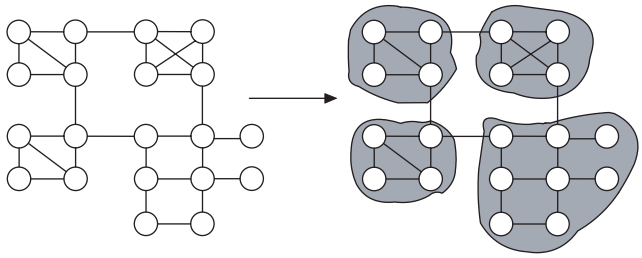

FIGURE 2. Decomposing a graph according to Λ value maximization.

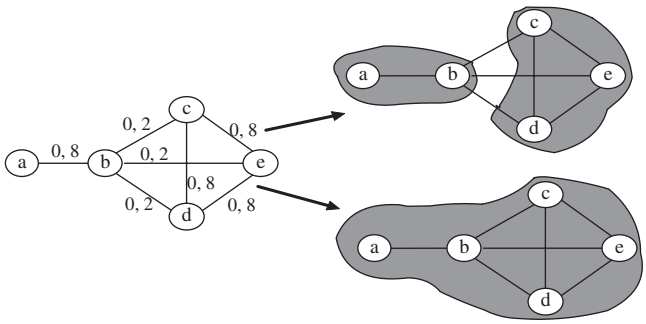

FIGURE 3. Clustering weighted graphs with WmajorClust.

$$\Lambda(C) = \sum_{i=1}^{n} |V(C_i)| \cdot w\lambda(C_i),$$

where $w\lambda(C_i)$ is the weighted edge connectivity of cluster C_i defined as the minimum sum of weights of edges E' that need to be removed in order to make C_i a non-connected graph. More specifically,

$$w\lambda(C_i) = \min \left\{ \sum_{e \in E'} w(e) : E' \subset E(C_i) \text{ and } C_i' = < V(C_i), E(C_i) \backslash E' > \text{ is not connected} \right\}$$

The maximization of $\Lambda(C)$ produces decomposition such that the connectivity between two vertices assigned to the same cluster is higher than the connectivity between any two vertices from two different clusters. Figure 2 presents an example of decomposing a graph with all edge weights equal to 1, according to Λ value maximization.

Figure 3 presents an example where the clustering behavior of WMajorClust is diversified in comparison with the clustering behavior of MajorClust. In Figure 3 the input graph is clustered in one cluster if we follow the MajorClust [20], whereas the WMajorClust optimizes the weighted edge connectivity and clusters the graph in two clusters. Although node (b) is connected to three nodes (c,d,e) which constitute a cluster, it is placed in another cluster with node (a) since the sum of weights of the edges (b,c), (b,d) and (b,e) is 0.6, whereas the weight of edge

TABLE 4. The clustering plagiarism detection algorithm WMajorClust.

```
void wMajorClust(Graph G) {
 int n=0; boolean t=false;
 for each vertex V in G V.clusterId= n++;
 while (!t) {
 t=true;
 for each vertex V in G {
  int clusterNo=maxNeibCluster(V);
  if (V.clusterId!=clusterNo) {
  V.clusterId=clusterNo;
  t=false;
 } } } }
```

```
int maxNeibCluster(Vertex v) {
 float weights[];
 for each edge E connected to V {
  Vertex neib=null;
  if (V.vertexId=E.vertexId1) neib=Vertex(E.vertexId2);
  else neib=Vertex(E.vertexId1);
  weights[neib.clusterId]+ =E.weight;
 }
 return maxPos(weights);
 }
```

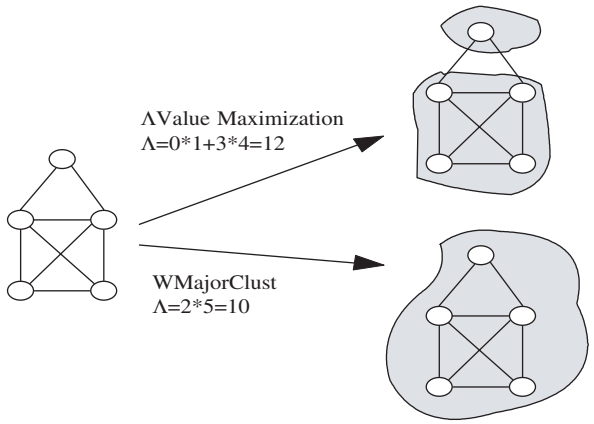

FIGURE 4. Suboptimal decomposition by WMajorClust.

(a,b) is 0.8. Thus, we may consider that edges (b,c), (b,d) and (b,e) have been eliminated by clustering. Since these edges represent similarity between programs, their elimination facilitates in some cases the discovery of false detections.

Table 4 presents WMajorClust. The algorithm initially assigns a cluster to each node of a graph and proceeds to merge nodes to clusters according to the weights of their edges. The algorithm terminates when no further merging is possible.

As shown in Table 4, each edge of graph G with vertex set $V(G)$ and edge set $E(G)$ is checked twice within the while-loop, i.e. once for each vertex it connects. In each iteration at least one node is reassigned a cluster ID and at least one cluster is enlarged. This continues until no node changes cluster ID. Thus, if $C_{\max}$, $V(C_{\max}) \subseteq V(G)$ designates a maximum cluster, i.e. the cluster with the maximum number of vertices from the clusters that are produced until the loop ends, then the time complexity of the algorithm is $\Theta(|E(G)| * |V(C_{\max})|)$.

Although WMajorClust, similar to MajorClust, approximates effectively the Λ value maximization decomposition, it does not always find the optimum solution as it disregards global criteria and it is restricted to local decisions (in every step only one node's neighbors are considered). Thus WMajorClust may lead to suboptimum Λ values. Figure 4 demonstrates such a case.

3. PERFORMANCE MEASURES FOR PLAGIARISM DETECTION

Our plagiarism detection reliability measure is based on precision and recall. Taking into account that PDetect detects as plagiarized all program pairs with a similarity value between the highest calculated value and a specified cutoff criterion value,

- *Precision* represents the proportion of actual plagiarism pairs that have been detected in the program pairs detected overall.
- *Recall* represents the proportion of actual plagiarism pairs that have been detected in the total set of program pairs that are actually plagiarized.

More formally, let us suppose we have a dataset with N program pairs and that the plagiarism detection algorithm labels program pairs either as plagiarized transcripts (positives) or as originals (negatives). A human reviewer also examines program pairs and estimates for each one of them if it is a plagiarism transcript or not. Thus, each pair in the test set is finally labeled as true positive, false positive, true negative or false negative. If TP is the total true positives, FP the total false positives, TN the total true negatives and FN the total false negatives, $N = \text{TP} + \text{FP} + \text{TN} + \text{FN}$ and precision and recall are defined as follows [22]:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Precision and recall may be combined to produce a total measure for plagiarism detection performance. According to [11], a more severe error is to lose plagiarism pairs having characterized them as negatives (false negative) than to characterize non-plagiarism pairs as positives (false positive). This is a reasonable hypothesis as the loss of false negatives introduces more work for the final evaluation by the human reviewer. Consequently, a total measure for plagiarism detection performance should be defined as a weighted sum of precision and recall.

DEFINITION 1. *The total plagiarism detection performance (TPDP) measure involves both precision and recall; it uses a parameter m, which indicates the importance of recall against precision for a particular task of plagiarism detection*

and it is defined as follows:

$$TPDP = P + m * R, \quad m \geq 1$$

Values of precision and recall vary according to the cutoff criterion value. Consequently, TPDP value varies according to the cutoff criterion value. Besides, cutoff criterion values that have been defined on the outputs of different plagiarism detectors are not comparable as they refer to different similarity measures. This variation makes impossible the comparison of different plagiarism detectors. However, outputs of different plagiarism detectors could be considered comparable at the cutoff criterion value where each plagiarism detector maximizes its TPDP.

DEFINITION 2. *The Optimum cutoff criterion (OCC) is defined as the cutoff criterion value at which TPDP is maximized.*

As the above definition implies, OCC is calculated by first calculating TPDP at every possible cutoff criterion value, which are all of the similarity values that are produced by a plagiarism detector. Then we choose the cutoff criterion value, which maximizes TPDP. Since these similarity values are sorted by the descending order, it is easy to compute TPDP at each one of them, as all pairs above it are considered positives and the rest are considered negatives. TP, FP, TN, FN may be counted as the similarity values are being processed. Thus, if n is the number of programs in the test set, then all possible similarity values are $[n * (n-1)]/2$ and the complexity of evaluating OCC is $\Theta(n^2)$.

4. CLUSTER ANALYSIS

In phase 1 results, programs are associated by a similarity value calculated by phase 1 processing. If we consider that a program is plagiarized from another if and only if they both belong to the same cluster, clustering may produce a new set of programs associations. This post-clustering set of program associations may be better or worse in terms of TPDP. Events that take place during clustering and cause a decrease in post-clustering TPDP are indicative of a bad quality of the produced clusters, whereas events that cause an increase in post-clustering TPDP are indicative of a good quality of the produced clusters.

4.1. Data structures definitions

- Let TS $= \{p_1, \ldots, p_n\}$ be a test set that includes n program sources with identifications $p_1, \ldots, p_n$.
- Let us define a similarity matrix $S_{n\times n} = (s_{ij})$, $0 \leq s_{ij} \leq 1$, where s_{ij} represents the similarity between programs $p_i, p_j \in$ TS. Output of plagiarism detectors is of the form of a similarity matrix S or it may be easily transformed to that form. However, for PDetect this is the format of the results of phase 1 processing.
- The post-clustering matrix PC is defined as $PC_{n\times n} = (pc_{ij})$, where $pc_{ij} = 1$ if program p_i belongs to the same cluster with program p_j, otherwise $pc_{ij} = 0$. PC is calculated after clustering.
- The human reviewer matrix HR is defined as $HR_{n\times n} = (hr_{ij})$, where $hr_{ij} = 1$ if human reviewer estimates programs $p_i, p_j \in$ TS are plagiarized from one another, otherwise $hr_{ij} = 0$.

A program pair $p_i, p_j \in$ TS is characterized under a specified cutoff criterion value (cc) as true positive, true negative, false positive and false negative, as follows:

	Before clustering	After clustering
True positive	$s_{ij} \geq$ cc, $hr_{ij} = 1$	$pc_{ij} = 1, hr_{ij} = 1$
True negative	$s_{ij} <$ cc, $hr_{ij} = 0$	$pc_{ij} = 0, hr_{\mathrm{ij}} = 0$
False positive	$s_{ij} \geq$ cc, $hr_{ij} = 0$	$pc_{ij} = 1, hr_{ij} = 0$
False negative	$s_{ij} <$ cc, $hr_{ij} = 1$	$pc_{ij} = 0, hr_{ij} = 1$

4.2. The clustering events

The following events occur when clustering a set of correlated programs and they should be identified since they affect the TPDP:

- Warning event 1: *Precision and recall increase*. A program pair $p_i, p_j \in$ TS is false negative before clustering and becomes true positive after clustering.
- Warning event 2: *Precision increase*. A program pair $p_i, p_j \in$ TS is false positive before clustering and becomes true negative after clustering.
- Warning event 3: *Precision and recall decrease*. A program pair $p_i, p_j \in$ TS is true positive before clustering and becomes false negative after clustering.
- Warning event 4: *Precision decrease*. A program pair $p_i, p_j \in$ TS is true negative before clustering and becomes false positive after clustering.

PDetect identifies these types of events and provides the user with suitable warnings. Events 1 and 2 are indicative of a good quality of the produced clusters, whereas events 3 and 4 are indicative of a bad quality of the produced clusters.

4.3. Human reviewer consistency

The consistency of human reviewer matrix should also be checked. Let us suppose three programs $p_i, p_j, p_k \in$ TS are placed in the same cluster. If pairs p_i, p_j and p_i, p_k are recognized by the human reviewer as pairs of plagiarism, it is expected that pair p_j, p_k is also a pair of plagiarism. The opposite is theoretically possible, if p_i contains a source segment plagiarized from p_j and another source segment plagiarized from p_k but p_j is not related to p_k. However this is a rather rare case and therefore, it is expected that corresponding entries in the human reviewer matrix is set equal to 1, $hr_{ij} = hr_{ik} = hr_{jk} = 1$. If one of them is set to 0, let us say $h_{jk} = 0$, then program pair p_j, p_k is supposed to be a false positive produced by phase 1 processing or introduced as a warning event 4 (*Precision decrease*). However, the fact that $h_{ij} = h_{ik} = 1$, makes it probable that h_{jk} value is incorrect and pair p_j, p_k should be true positive. However, if only one of the $hr_{ij}, hr_{ik}, hr_{jk}$ values is set to 1, let us say $hr_{ij} = 1$ and $hr_{ik} = hr_{jk} = 0$, we do not have a basis to assume inconsistency in human reviewer matrix. Thus, program

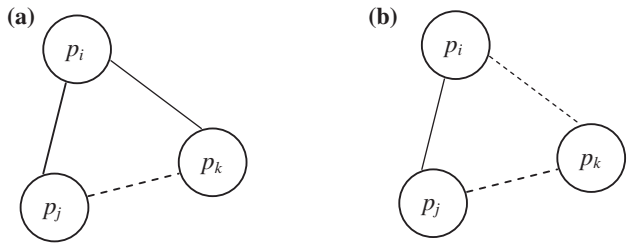

FIGURE 5. Inconsistencies in Human Reviewer Matrix. **(a)** Possible human reviewer error (p_j, p_k). Discontinuous lines correspond to HR values of 0. Continuous lines correspond to HR values of 1. **(b)** Possible bad member of the cluster, p_k.

pairs p_i, p_k and p_j, p_k are false positives and therefore, program p_k is probably a bad member of the cluster. For every triple of programs p_i, p_j, p_k that belong to the same cluster, PDetect identifies the following warning events:

- Warning event 5: *Possible human reviewer error*, if $hr_{ij} + hr_{ik} + hr_{jk} = 2$.
- Warning event 6: *Possible bad member of the cluster*, if $hr_{ij} + hr_{ik} + hr_{jk} = 1$.

Both cases are demonstrated in Figure 5.

5. SOURCE CODE SETS AND EXPERIMENTATION

We choose to evaluate PDetect in comparison with JPlag, since [11] claims that JPlag is more reliable than MOSS (in terms of plagiarism detection performance) and we assume that it is at least as reliable as YAP3 (since JPlag is an improvement of YAP3). Two tests were devised for this evaluation. One of them is based on two sets of actual student programming assignments. The other test is based on an artificial work set devised so as to 'cheat' PDetect and JPlag. Both tests were carried out using the default minimum-match-length parameter for JPlag and default weights for PDetect keywords.

5.1. The real datasets

Two sets (Available from http://iiu.teikav.edu.gr/users/lmous/PDetect.htm) of real C++ programming assignments have been used for this comparative evaluation.

- The 'Calculator' assignment requires advanced programming skills and detailed knowledge of C++. A total of 24 programs were submitted, with an average program length of 257 lines and an average use of 95 keywords per program.
- The 'Snakes' assignment was comparatively easier to program. A set of 51 programs were submitted, with an average program length of 178 lines and an average use of 70 keywords per program.

5.1.1. *Evaluation procedure*

(i) PDetect phase 1 processing takes place for both test sets. JPlag is also executed with the same input. Corresponding similarity matrices are produced.
(ii) The human reviewer evaluates both test sets and produces corresponding human reviewer matrices.
(iii) The clustering algorithm is invoked for PDetect phase 1 output and for JPlag output. Clustering takes place at the corresponding OCC.[3] A list of warning events together with a post-clustering similarity matrix is produced for each test set.
(iv) Steps ii and iii are repeated, assisted by the lists of warning events until the human reviewer ensures the reliable evaluation of both test sets.
(v) Combined results of PDetect phase 1 output and JPlag output are produced based on the combined similarity matrix *CS*. Let $SPD_{n\times n} = (spd_{ij})$ be the similarity matrix produced by PDetect phase 1 results; OCCPD is the corresponding OCC. Similar to *SPD* is the Similarity Matrix $SJP_{n\times n} = (sjp_{ij})$ produced by JPlag output and OCCJP is the corresponding OCC. Then the combined similarity matrix *CS* is defined as $CS_{n\times n} = (cs_{ij})$, where $cs_{ij} = 1$, if $spd_{ij} \geq$ OCCPD or $sjp_{ij} \geq$ OCCJP, otherwise $cs_{ij} = 0$.

5.1.2. *Results*

Results are summarized in the Tables 5 and 6. In Table 5 TPDP values are presented for phase 1, post-clustering and combined results.

- *PDetect phase 1—JPlag:* Phase 1 TPDP is higher for PDetect in both assignments. Although precision is higher for JPlag in both cases, the difference in recall results in a higher TPDP for PDetect.
- *Phase 1 TPDP—post-clustering TPDP:* Post-clustering TPDP is better in comparison with phase 1 TPDP for JPlag in both assignments. The high precision in the phase 1 result remains unchanged in post-clustering results and reflects the absence of false positives. This absence together with the better post-clustering recall indicates the good quality of the produced clusters at that level of post-clustering TPDP. However, in assignment 'Calculator' PDetect presents a decrease in post-clustering TPDP, which is caused by a major decrease in the precision. Thus, false positives have been introduced during clustering and the quality of the produced clusters is questionable. In assignment 'Snakes' TPDP remains constant and the quality of the produced clusters is expected to be reasonably good at that level of post-clustering TPDP.
- *Combined results:* In both assignments, combined TPDP is better when compared with TPDP of phase 1

[3] Although experimentation has taken place using both variants of TPDP $(P + 2R, P + 3R)$, we present only the results based on the $P + 2R$ variant. This decision is based on the fact that there is no difference in the final conclusions produced by the different variants of TPDP. Thus, in the following when we are referring to TPDP we mean the $P + 2R$ variant and when referring to OCC we mean the one that is calculated based on $P + 2R$.

TABLE 5. Experimental data TPDP values.

	Snakes				Calculator			
	OCC	Precision	Recall	TPDP	OCC	Precision	Recall	TPDP
Phase 1								
PDetect	0.714	0.826	0.950	2.726	0.573	0.939	0.958	2.855
JPlag	0.482	1.000	0.632	2.263	0.572	1.000	0.574	2.149
Post-clustering								
PDetect	1.000	0.826	0.950	2.726	1.000	0.527	1.000	2.527
JPlag	1.000	1.000	0.684	2.368	1.000	1.000	0.787	2.574
Combined	1.000	0.792	1.000	2.792	1.000	0.939	0.979	2.896

TABLE 6. Clusters of experimental data.

	Snakes
Human reviewer	{(0,1,4), (2,3,11), (5,6,9,10), (7,8,12), (13,14), (15,16), (18,19), (20,21), (26,40)}
PDetect	{(0,1,4), (2,3,11), (5,6,9,10,17), (7,8,12), (13,14), (15,16), (18,19), (20,21)}
JPlag	{(0,1,4), (2,3,11), (5,10,9), (13,14), (15,16), (18,19), (26,40)}
	Calculator
Human reviewer	{(0,1,2,3,4,5,6,7,11,12), (8,9,10)}
PDetect	{(0,1,2,3,4,5,6,7,8,9,10,11,12,13)}
JPlag	{(0,1,2,4,5,3,7,6,11), (8,9)}

in both JPlag and PDetect. This finding reveals that PDetect and JPlag are sensitive to different types of attacks.

In Table 6 the numbers correspond to program identifications. The human reviewer line presents the actual clusters of plagiarism, which have been calculated, based on the human reviewer matrix; PDetect clusters have been calculated based on PDetect phase 1 similarity matrix and JPlag clusters have been calculated based on JPlag similarity matrix. In assignment 'Snake,' PDetect omits cluster (26,40) and adds false member 17 in cluster 3. This result is accompanied by an appropriate warning: '*Possible bad member of the cluster. Cluster ID: 3, member ID: 17*' and it is reasonable if taking into account the corresponding TPDP values. In the same assignment, JPlag omits clusters (7,8,12), (20,21) and member 6 from cluster 3. Again this result is in accordance with the corresponding TPDP values. In assignment 'Calculator,' PDetect has merged the two clusters into one and has added false member 13. This result is also accompanied by a corresponding list of warning events of type 4 (*precision decrease*) and 6 (*possible bad member of the cluster*). JPlag produces two clusters but omits member 12 from cluster 1 and member 10 from cluster 2.

5.2. The artificial datasets

Two plagiarized transcripts were produced based on an original C++ program called the 'original'. One of the plagiarized transcripts is named 'JPlagConfusion' and the other 'PDetectConfusion'. 'JPlagConfusion' encapsulates modifications of the type that maximizes the loss of JPlag detection performance. More precisely, all possible code shuffling has been performed but no structural modification has been done. 'PDetectConfusion' encapsulates all possible modifications to keywords that are modifications of the type that confuses PDetect. Again no structural modification has been performed. In both cases, code has not been removed or added. Table 7 demonstrates the two types of modifications.

These three programs constitute the test set that was given to both PDetect and JPlag in order to calculate the pairwise similarities. The average program length is 193 lines and the average use of keywords per program is 137. Table 8 presents the results.

As shown in Table 8, JPlag 'loses' the pair original-JPlagConfusion, whereas PDetect calculates for this pair a similarity value equal to 1. Although the similarity values that are calculated for pair original-PDetectConfusion by PDetect (0.97) and by JPlag (0.94) are not comparable, as they do not use the same similarity measure and therefore we cannot say which one performs better, it is obvious because of the high similarity value that has been calculated by PDetect that it has not been cheated. This may be explained by the fact that only a few keywords may be modified in a program with no structural modifications, additions or deletions of code. Finally, we observe that the confusion of JPlag has been transmitted in pair JPlagConfusion–PDetectConfusion. This is expected, as code has not been shuffled in PDetectConfusion transcript.

5.3. Interpretation of the results

In the following paragraphs, we discuss the results of the experimentation, based on their comparative summary as presented in Figure 6.

PDetect achieves a higher TPDP value on phase 1. In both sets of real data, PDetect has achieved a higher TPDP before clustering. This result is because of the representation of programs as indexed sets of substitute keywords. Keywords compose the fundamental structure of each program. Thus, distribution of keywords in a program reflects its structure. Modifications on a program that do not alter its structure do not also alter its distribution of keywords.

TABLE 7. Examples of the two types of code modifications that affect JPlag and PDetect TPDP.

Original	JPlagConfusion	pDetectConfusion
int sz,lastEntry,keyIterator;	**bool** sorted;	Same as in original
bool sorted;	**int** sz,lastEntry,keyIterator;	
int apo=0,	Same as in original	**int** apo=0 ;
mexri=sz,		**int** mexri=sz ;
cIdx;		**int** cIdx;
if (pos== −1) **return** "";	Same as in original	if (pos== −1) **return** "";
else return store[pos].data;		**return** store[pos].data;

TABLE 8. Calculated similarity for artificial programs set.

Program pairs		Similarity	
		PDetect	JPlag
Original	JPlagConfusion	1	0.3
Original	pDetectConfusion	0.97	0.94
JPlagConfusion	pDetectConfusion	0.97	0.25

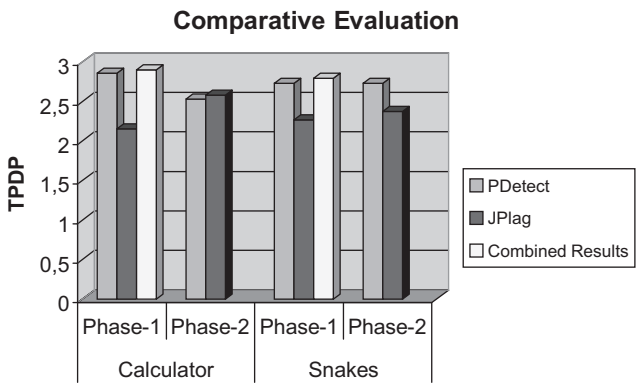

FIGURE 6. Experimentation results based on comparative summary.

In addition, it seems quite difficult to alter the distribution of keyword in some programs without proceeding to major code modifications. From another point of view, JPlag (as other structure metric systems) 'suffers' from what is called local confusion [11]. Local confusion occurs when code segments shorter than the minimum-match-length parameter have been shuffled. For example, if we compare functions 'init' and 'initGame' in Table 1, JPlag with its default minimum-match-length will calculate a similarity value of 0 because of local confusion. Local confusion is also the cause for which JPlag calculates such a low similarity value between 'original' and 'JplagConfusion' in the test with the artificial programs set. Overall, pDetect uses a global measure and does not suffer from local confusion, since there is no kind of code shuffling capable of tricking the system.

Clustering affects TPDP. As has already been discussed, clustering affects TPDP either by associating programs as being members of the same cluster or by dissociating them as being members of different clusters. The corresponding events between these modifications of program associations may constitute a valuable assist to the human reviewer. It should also be noted that clustering is easily encapsulated in any plagiarism detection system and it is not restricted to programming assignment plagiarism detectors. This is owing to the fact that clustering is not related to the actual structure of the objects that are clustered and therefore, it may be performed on any set of correlated objects.

Combined results are beneficial. As has already reported, the failure of JPlag is because of local confusion. From another point of view, empirical observation shows that PDetect failure may occur when some parts of a program are plagiarized and other are original. Thus, the two systems usually fail for different types of attacks and for this reason they may complement each other.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a process of two phases for detecting plagiarism in students' programming assignments. Although the phase 1 process is actually an attribute-counting system, it is quite efficient when compared with structure-metric systems. Over the basic detection functionality we propose clustering, which benefits the instructional strategies, improves plagiarism detection performance and may be applied as a phase 2 processing to every plagiarism detector. In this context, we also propose an inspection approach for human reviewer consistency. Although our approach is characterized by its inability to visualize the result of the comparison of two sources, it may be used as complementary to structure metric systems since combined results exhibit better detection performance.

Future research efforts should focus on the following:

- Contact an extensive comparative evaluation between PDetect, JPlag and MOSS, which will be based on large datasets that we are currently collecting. In this comparative evaluation, we are also going to experiment with keyword weights.
- Automatically estimate a range for the OCC based on the average use of keywords per program, or taking into account the similarity distribution under examination, or utilizing some other factor of the system.
- Develop solutions to avoid false detections: program representation as indexed sets of substitute keywords

seems to be a suitable method to gain a high TPDP. However, TPDP is high mainly because of the high recall value. It seems that precision may be better if some other dimension were also to be considered in the representation of programs.

- Develop a framework to combine the functionality of PDetect with the functionality of a traditional structure metric system increasing in that way the TPDP.
- Develop a web-service[4] that would make PDetect publicly available is of great importance, since instructors may use it over the Web and it could also be tested by other researchers.
- Adapt TPDP measure to the notion of clusters. As the final results of PDetect consist of a set of clusters, an appropriate precision measure should count the number of real clusters of plagiarism in the number of clusters that have been detected. Also, an appropriate recall measure should count the number of clusters that have been detected as a portion of real clusters of plagiarism that exist in the input set.

ACKNOWLEDGEMENTS

The authors thank the anonymous referees for their valuable comments and advice, which have considerably improved the quality, presentation and readability of the paper.

REFERENCES

[1] TR 14-03 (2003) *An XML plagiarism detection model for procedural programming languages*. Iowa State University, IA.

[2] Halstead, M. (1977) *Elements of Software Science*. Elsevier, New York.

[3] Ottenstein, K. (1976) An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bull.*, **8**, 30–41.

[4] Grier, S. (1981) A tool that detects plagiarism in Pascal programs. *ACM SIGCSE Bull.*, **13**, 21–25.

[5] Donaldson, J., Lancaster, A. and Sposato, P. (1981) A plagiarism detection system. *ACM SIGCSE Bull.*, **13**, 15–20.

[6] Faidhi, J. and Robinson, S. (1987) An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Comput. Educ.*, **11**, 11–19.

[7] Allen, F. and Cocke, J. (1976) A program data flow analysis procedure. *Commun. ACM*, **19**, 137–147.

[8] Verco, K. and Wise, M. (1996) Software for detecting suspected plagiarism: comparing structure and attribute counting systems, In *Proc. First Australian Conf. on Computer Science Education*, Sydney Australia, July 3–5, pp. 86–95. ACM Press, New York, USA.

[9] Aiken, A. (1998) MOSS: a system for detecting software plagiarism. University of Berkeley, CA. Available at http://www.cs.berkeley.edu/∼aiken/moss.html.

[10] Wise, M. (1996) YAP3: improved detection of similarities in computer program and other text. In *Proc. 27th SIGCSE Technical Symp. on Computer Science Education*, Philadelphia USA, February 15–18, pp. 130–134. ACM Press, New York, USA.

[11] Prechelt, L., Malpohl, G. and Philippsen M. (2002) Finding plagiarisms among a set of programs with Jplag. *J. Univ. Comput. Sci.*, **8**, 1016–1038.

[12] Schleimer, S., Wikerson, D. and Aiken, A. (2003) Winnowing: local algorithms for document fingerprinting. In *Proc 2003 ACM SIGMOD Int. Conf. on Management of Data*, San Diego, CA, June 9–12, pp. 76–85. ACM Press, New York, USA.

[13] Wise, M. (1993) String similarity via greedy string tiling and running Karp-Rabin matching. Department of Computer Science, University of Sydney. Available at ftp://ftp.cs.su.oz.au/michaelw/doc/RKR_GST.ps.

[14] Ribler, R. and Abrams, M. (2000) Using visualization to detect plagiarism in computer science classes. In *Proc. IEEE Symp. on Information Visualization*, Salt Lake City, UT, October 9–10, pp. 173–178. IEEE Computer Society, Los Alamitos, CA.

[15] Hodges, C. (2004) Designing to motivate: motivational techniques to incorporate in e-learning experiences, *The Journal of Interactive Online Learning*, **2**(3), 1–7.

[16] McConnell, D. (2000) *Implementing Computer Supported Cooperative Learning*. Kogan Page, London, UK.

[17] Parker, A. and Hamblen, J. (1989) Computer algorithms for plagiarism detection. *IEEE Trans. on Educ.*, **32**, 94–99.

[18] Chakrabarti, S. (2003) *Mining the Web.*, Morgan Kaufmann Publishers, Oxford.

[19] Lee, L. (1999) Measures of distributional similarity. In *Proc. 37th Annual Meeting of the Association for Computational Linguistics*, College Park, Md, June 22–27, pp. 25–32. ACL, East Stroudsburg, PA.

[20] Stein, B. and Niggemann, O. (1999) On the nature of structure and its identification. In *Proc. Graph-Theoretic Concepts in Computer Science: 25th Int. Workshop, WG'99*, Ascona, Switzerland, June 17–19, pp. 122–134. Springer-Verlag, Heidelberg.

[21] Chartrand, G. and Oellermann, O. (1993) *Applied and Algorithmic Graph Theory*. McGraw-Hill, Inc., New York.

[22] Hand, D., Mannila, H. and Smyth, P. (2001) *Principles of Data Mining*. Bradford Books, MIT Press, MA.

[4] Submission is currently performed by email. Visit http://iiu.teikav.edu.gr/users/lmous/PDetect.htm to submit programs for evaluation by PDetect.