GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis*

Chao Liu, Chen Chen, Jiawei Han
Department of Computer Science
University of Illinois-UC, Urbana, IL 61801
{chaoliu, cchen37, hanj}@cs.uiuc.edu

Philip S. Yu
IBM T. J. Watson Research Center
Hawthorne, NY 10532
psyu@us.ibm.com

ABSTRACT

Along with the blossom of open source projects comes the convenience for software plagiarism. A company, if less self-disciplined, may be tempted to plagiarize some open source projects for its own products. Although current plagiarism detection tools appear sufficient for academic use, they are nevertheless short for fighting against serious plagiarists. For example, disguises like statement reordering and code insertion can effectively confuse these tools. In this paper, we develop a new plagiarism detection tool, called GPLAG, which detects plagiarism by mining program dependence graphs (PDGs). A PDG is a graphic representation of the data and control dependencies within a procedure. Because PDGs are nearly invariant during plagiarism, GPLAG is more effective than state-of-the-art tools for plagiarism detection. In order to make GPLAG scalable to large programs, a statistical lossy filter is proposed to prune the plagiarism search space. Experiment study shows that GPLAG is both effective and efficient: It detects plagiarism that easily slips over existing tools, and it usually takes a few seconds to find (simulated) plagiarism in programs having thousands of lines of code.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications - Data Mining

General Terms

Algorithms, Experimentation

*This work was supported in part by the U.S. National Science Foundation NSF ITR-03-25603 and IIS-03-08215/05-13678. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies.

KDD'06, August 20–23, 2006, Philadelphia, Pennsylvania, USA.

Keywords

Program dependence graph, Graph mining, Software plagiarism detection

1. INTRODUCTION

Along with the blossom of open source projects comes the convenience for software plagiarism. Suppose a company needs to implement a large software product, which, if done from the scratch, could be time-consuming. The developers, if less self-disciplined, may be tempted to find a counterpart in some open source projects, rip off the interface components, like I/O and Graphical User Interfaces (GUIs), and finally fit the essential components (*i.e.*, core parts) into their own project *with serious disguises*. Because only core parts are plagiarized, which accounts for a small portion of the whole project, and heavy disguises are applied, the plagiarism, which we call *core-part plagiarism*, is hard to notice. In this paper, we study how to detect core-part plagiarism both accurately and efficiently. In the above example, the open source project from which code is copied is the *original program*, and the company's project is called a *plagiarism suspect*.

A quality plagiarism detector has strong impact to lawsuit prosecution. It reveals where are the plagiarized parts and what plagiarism operations are applied, which may otherwise be hard for human beings to identify due to the large code size and tricky disguises.

Although current plagiarism detection tools appear sufficient for academic use, like finding copied programs in programming classes, they are nevertheless short for fighting against serious plagiarists. These tools are mainly based on program token strings, which, as will be explained in Section 3.2, are fragile to some disguises that can be done automatically. For example, disguises like statement reordering, replacing a `while` loop with a `for` loop, and code insertion can effectively confuse these tools. Therefore, more robust detection algorithms are well needed.

From a knowledge discovery point of view, the detection of core-part plagiarism is actually an interesting data mining problem. In the first place, plagiarism detection is in essence to find from source code interesting patterns that uncover disguised code changes. These patterns should be an intrinsic representation of programs such that they are hardly overhauled in plagiarism. In the second place, because the plagiarized core parts only account for a small portion of the entire program, finding these real plagiarized parts is like anomaly detection. For example, the detec-

tion is expected to be low at false positive rate. Finally, for practical concern, the detection algorithm should scale to large programs in both accuracy and efficiency. These three factors altogether make core-part plagiarism detection a challenging, as well as interesting, data mining problem.

We examined those disguises that are effective in confusing current plagiarism detection tools, and found that they are nearly futile to the program dependence graph (PDG): PDGs almost stay the same even when the source code is significantly altered. A PDG is a graph representation of the source code of a procedure, where statements are represented by vertices, and data and control dependencies between statements by edges (details in Section 2). Intuitively, PDGs encode the program logic, and in turn reflect developers' thinking when code is written. Code changes regardless of dependencies are prone to errors, and a plagiarist who wants to alter PDGs through code changes should understand the program first. Thus, a plagiarist can freely modify the code, but as long as the program correctness is preserved, the dependence graph is hardly overhauled. Although an extraordinarily creative and diligent plagiarist *may* correctly overhaul the PDGs, the cost is likely higher than rewriting her own code, which contradicts with the incentive of plagiarism. After all, plagiarism aims at code reuse with disguises, which requires much less effort than writing one's own.

We develop a PDG-based plagiarism detection algorithm, which exploits the invariance property of PDGs. Suppose the original program $\mathcal{P}$ and the plagiarism suspect $\mathcal{P}'$ each have n and m procedures, then the two programs are represented by two PDG sets $\mathcal{G}$ and $\mathcal{G}'$, respectively, and $|\mathcal{G}| = n$ and $|\mathcal{G}'| = m$. Then the problem of plagiarism detection boils down to two sub-problems: First, given $g \in \mathcal{G}$ and $g' \in \mathcal{G}'$, how can we decide whether g' is a plagiarized PDG of g? Second, how to efficiently locate real plagiarized PDG pairs, while in principle $n * m$ pairs are to be checked?

We approach the first problem through relaxed subgraph isomorphism testing: Whenever g is γ-isomorphic to g', (g, g') is regarded as a plagiarized PDG pair, where γ is the relaxation parameter. Although subgraph isomorphism testing is in general NP-complete, it is totally tractable in this application. In the first place, PDGs cannot be arbitrarily large as procedures are designed to be of reasonable size for developers to manage. Secondly, PDGs are not general graphs, and their peculiarity, like varieties of vertex types, makes backtrack-based isomorphism algorithm efficient. Finally, different from conventional isomorphism testing, we are satisfied as long as one, rather than *all*, isomorphism between g and g' is found. These three factors make isomorphism testing efficient, although it appears formidable at the first glance.

As to the second problem, we notice that only a small portion of the entire $n * m$ PDGs pairs need isomorphism testing. Most PDGs pairs can be excluded from detailed isomorphism testing because they are dissimilar even with a high-level examination. Therefore, we design a lossy filter to prune these dissimilar PDG pairs. Different from conventional similarity measurement that usually is based on a certain distance metric, this filter follows a similar reasoning to hypothesis testing: A PDG pair (g, g') is preserved until enough evidence is collected against the similarity between g and g'. In comparison with distance-based methods, this approach avoids the difficulty of proper parameter setting, and it also provides a statistical estimation of the *false negative* rate. In experiments, the lossy filter, collaborating with another lossless filter, usually prunes about nine tenths of the original search space.

Based on the above design, we implemented a PDG-based plagiarism detection tool, called GPLAG. Experimental results indicate that GPLAG is both effective and efficient: It accurately catches plagiarism that slips over current state-of-the-art detection tools, and it takes a few seconds to find (simulated) core-part plagiarism in programs having thousands of lines of code.

In summary, we makes the following contributions in this study:

1. We design and implement a PDG-based plagiarism detection tool, called GPLAG, which conducts plagiarism analysis on the program dependence graphs. Because PDGs are robust to the disguises that confuse current state-of-the-art tools, GPLAG is more effective, and hence more suitable for industrial use.
2. In order to make GPLAG scalable to large programs, and suitable for core-part plagiarism detection, we design a statistical lossy filter, which, when collaborating with a lossless filter, significantly prunes the search space. This makes GPLAG efficient for large programs.
3. Finally, this study introduces PDGs, a new kind of graphs, to the data mining community. Graphs have been used for data modeling in many domains. Here PDGs are introduced as a graphic modeling for program source code. This paper exemplifies that proper mining of PDGs can lead to more effective plagiarism detection algorithm.

The rest of the paper is organized as follows. Section 2 introduces the program dependence graph and related graph terminologies. We review previous plagiarism detection techniques in Section 3. The details of PDG-based plagiarism detection and its implementation are discussed in Sections 4 and 5. The experimental evaluations are presented in Section 6. Section 7 discusses the related work and the potential implications of the GPLAG approach to software industry. Finally, Section 8 concludes this study.

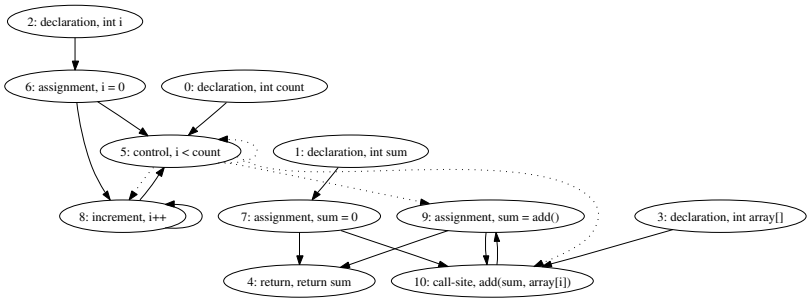
2. BACKGROUND

A *program dependence graph* (PDG) is a graph representation of the source code of a procedure [4]. Basic statements, like variable declarations, assignments, and procedure calls, are represented by program vertices in PDGs. Each vertex has one and only one type, and several important types are listed in Table 1, which also illustrates how source code is decomposed and mapped to program vertices. The data and control dependencies between statements are represented by edges between program vertices in PDGs.

DEFINITION 1 (CONTROL DEPENDENCY EDGE). *There is a control dependency edge from a "control" vertex to a second program vertex if the truth of the condition controls whether the second vertex will be executed.*

DEFINITION 2 (DATA DEPENDENCY EDGE). *There is a data dependency edge from program vertex v_1 to v_2 if there is some variable* `var` *such that:*

- *v_1 may be assigned to* `var`*, either directly or indirectly through pointers.*
- *v_2 may use the value in* `var`*, either directly or indirectly through pointers.*



(a) Program Dependence Graph of the Procedure `sum`

```
int sum(int array[], int count)
{
  int i, sum;
  sum = 0;
  for(i = 0; i < count; i++){
    sum = add(sum, array[i]);
  }
  return sum;
}

int add(int a, int b)
{
  return a + b;
}
```

(b) Summation over an Array

Figure 1: An Illustrative Example for Program Dependence Graphs

- *There is an execution path in the program from the code corresponding to v_1 to the code corresponding to v_2 along which there is no assignment to* `var`.

Definition 3 (Program Dependence Graph). *The program dependence graph G for a procedure P is a 4-tuple element $G = (V, E, \mu, \delta)$, where*

- *V is the set of program vertices in P*
- *$E \subseteq V \times V$ is the set of dependency edges, and $|G| = |V|$*
- *$\mu : V \to S$ is a function assigning types to program vertices,*
- *$\delta : E \to T$ is a function assigning dependency types, either data or control, to edges.*

Therefore, a program dependence graph is a directed, labelled graph, which represents the data and control dependencies *within* one procedure. It depicts how the data flows between statements, and how statements control or are controlled by other statements.

Figure 1 provides an example to illustrate program dependence graph. Figure 1(a) depicts the PDG of the procedure `sum` whose code is on the right in Figure 1(b). Data and control dependencies are plotted in solid and dashed lines respectively. Specifically, the text inside each vertex gives its vertex id, vertex type, and corresponding source code. The edges are explained by Definitions 1 and 2.

Because we will use graph isomorphism to detect plagiarism, related terminologies are defined below.

Definition 4 (Graph Isomorphism). *A bijective function $f : V \to V'$ is a graph isomorphism from a graph $G = (V, E, \mu, \delta)$ to a graph $G' = (V', E', \mu', \delta')$ if*

- *$\mu(v) = \mu'(f(v))$,*
- *$\forall e = (v_1, v_2) \in E, \exists e' = (f(v_1), f(v_2)) \in E'$ such that $\delta(e) = \delta(e')$,*
- *$\forall e' = (v_1', v_2') \in E', \exists e = (f^{-1}(v_1'), f^{-1}(v_2')) \in E$ such that $\delta(e') = \delta(e)$*

Definition 5 (Subgraph Isomorphism). *An injective function $f : V \to V'$ is a subgraph isomorphism from G to G' if there exists a subgraph $S \subset G'$ such that f is a graph isomorphism from G to S.*

Definition 6 (γ-Isomorphic). *A graph G is γ-isomorphic to G' if there exists a subgraph $S \subseteq G$ such that S is subgraph isomorphic to G', and $|S| \geq \gamma|G|$, $\gamma \in (0, 1]$.*

Type	Description
call-site	Call to procedures.
control	If, switch, while, do-while, or for.
declaration	Declaration for a variable or formal parameter.
assignment	Assignment expression.
increment	++ or −− expression
return	Function return expression.
expression	General expression except the above three, like one with ? operator
jump	Goto, break, or continue
label	Program labels
switch-case	Case or Default

Table 1: Program Vertex Types

3. SOFTWARE PLAGIARISM DETECTION

This section reviews existing plagiarism detection algorithms. We first illustrate common plagiarism disguises through an example in Section 3.1, and then analyze the shortcomings of existing techniques for plagiarism detection in Section 3.2.

3.1 Plagiarism Disguises

Figure 2 shows an example of plagiarism. The left procedure `make_blank` is the original code, and is excerpted from a program `join`. This `join` program joins lines of two files on a common field, and it is shipped with all Linux and Unix distributions. It has 667 lines of C code, excluding blanks and comments, and the procedure `make_blank` is one of the entire 17 procedures. On the right is a plagiarized version of the procedure, prepared by the authors. It exemplifies typical disguises that are commonly employed in plagiarism. From trivial to complicated, they are

1. Format alteration (FA): Insert and remove blanks and/or comments.

2. Identifier Renaming (IR): Identifier names can be consistently changed without violating program correctness. For example, in Figure 2, the procedure name `make_blank` is changed to `fill_content`, variable `blank` is changed to `fill`, `buf` to `store`, etc. Identifier renaming can confuse human beings, but is almost futile to detection tools.

3. Statement reordering (SR): Some statements can be reordered without causing program errors. For example, the

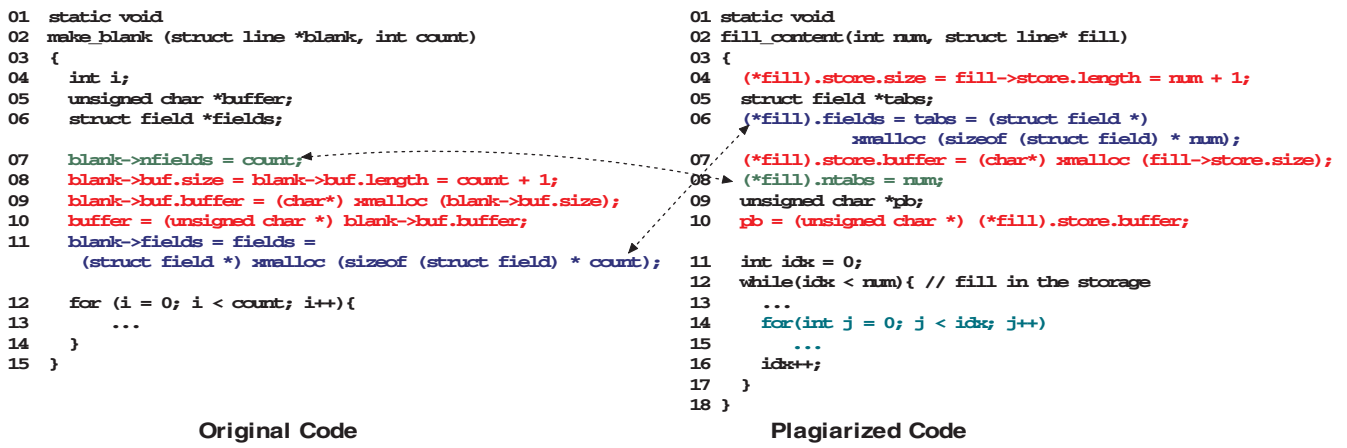


Figure 2: Original and Plagiarized Code

three declarations at lines 4 to 6 in the original code are reordered and scattered in the plagiarized code. Moreover, the statements on lines 7 and 11 are also reordered, as indicated by the dotted lines. In contrast, the statements from lines 8 to 10 cannot be reordered due to their sequential dependencies.

4. Control Replacement (CR): A `for` loop can be equivalently replaced by a `while` loop, or by an infinite `while` loop with a `break` statement, and vice versa. An `if(a){A}else{B}` block can be replaced by `if(!a){B}else{A}` for the same logic. In Figure 2, the `for` loop from lines 12 to 14 in the original code is replaced by a `while` loop on the right.

5. Code Insertion (CI): Immaterial code can be inserted to disguise plagiarism, provided the inserted code does not interfere with the original program logic. In the plagiarized code, a `for` loop is inserted at lines 14 and 15.

3.2 Review of Plagiarism Detection

We now review existing techniques for plagiarism detection and examine how each of them is robust to the above five kinds of disguises. Roughly, these techniques fall into the following three categories.

1. String-based: Each statement is treated as a string, and a program is represented as a sequence of strings. Two programs are compared to find sequences of same strings [1]. Because blanks and comments are discarded, this kind of algorithms are robust to format alteration, but fragile to identifier renaming.

2. AST-based: A program is first parsed into an abstract syntax tree (AST) with variable names and literal values discarded. Then duplicate subtrees are searched between two programs, and code corresponding to duplicate subtrees are labelled as plagiarism [2, 11]. Because this approach disregards the information about variables (in order to make codes differing on variables names appear the same on ASTs), it ignores data flows, and is in consequence fragile to statement reordering. In addition, it is also fragile to control replacement.

3. Token-based: In this approach, program symbols, like identifiers and keywords, are first tokenized. A program is then represented as a token sequence, and duplicate token subsequences are searched for plagiarism between two programs [9, 17, 18]. Because variables of the same type are mapped into the same token, this approach is robust to identifier renaming. However, since this approach relies on sequential analysis, it is generally fragile to statement reordering, and code insertion: A reordered or inserted statement can break a token sequence which may otherwise be regarded as duplicate to another sequence. This fragility can be partially remedied through fingerprinting [18], but it does not fundamentally save token-based algorithms. Finally, token-based methods are also fragile to control replacement because `for` and `while` loops render different token sequences: Not only are the keywords changed, but the code for iteration indexing and condition checking is also moved around.

Two representatives of token-based algorithms, MOSS [18] and JPLAG [17], are the two most commonly used tools for plagiarism detection in practice. They prove effective in detecting plagiarism in programming classes. However, since they are token-based, experienced plagiarist can confuse them through disguises, such as statement reordering and code insertion. Therefore, an approach that is robust (at least) to the five kinds of disguises are expected, and PDG-based algorithms turn out to be a suitable choice. In summary, the robustness of algorithms based on different representation of programs is compared in Table 2.

	String	AST	Token	PDG
Format Alt.	Yes	Yes	Yes	Yes
Id Rename	No	Yes	Yes	Yes
Stmt Reorder	No	No	Partial	Yes
Ctrl Replace	No	No	Partial	Yes
Code Insert	No	No	No	Yes

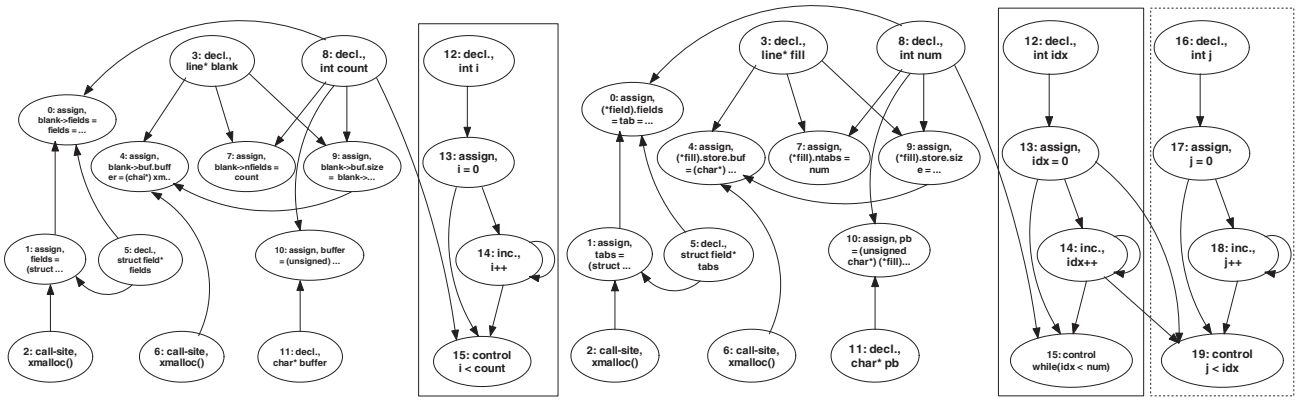
Table 2: Robustness Comparison

4. PDG-BASED PLAGIARISM DETECTION

In this section, we discuss PDG-based plagiarism detection. We first formulate the problem in Section 4.1, where we decompose the problem of plagiarism detection into two sub-problems. We then address the two sub-problems in Sections 4.2 and 4.3, respectively. Finally, Section 4.4 discusses the computation feasibility.

4.1 Problem Formulation

Given an original program $\mathcal{P}$, and a plagiarism suspect $\mathcal{P}'$, plagiarism detection tries to search for duplicate structures



(a) PDG of the Original Code (b) PDG of the Plagiarized Code

Figure 3: Program Dependence Graphs of the Original and Plagiarized Code in Figure 2

between $\mathcal{P}$ and $\mathcal{P}'$ in order to prove or disprove the existence of plagiarism. By representing a program as a set of PDGs, the search for duplicates are performed on PDGs.

Suppose $\mathcal{P}$ and $\mathcal{P}'$ each have n and m procedures, then they are represented by two PDG sets $\mathcal{G}$ and $\mathcal{G}'$ respectively, and $|\mathcal{G}| = n$ and $|\mathcal{G}'| = m$. Then the problem of plagiarism detection consists of the following two sub-problems:
1. Given $g \in \mathcal{G}$ and $g' \in \mathcal{G}$, how to judge whether the corresponding procedure of g' is plagiarized from that of g? If it is, (g, g') is called a *plagiarized pair*, or a *match* for short.
2. Given $\mathcal{G}$ and $\mathcal{G}'$, how to locate most, if not all, plagiarism pairs both accurately and efficiently?

Since we are in particular interested in detecting core-part plagiarism, where only a small number of procedures in $\mathcal{P}'$ are plagiarized from $\mathcal{P}$ if plagiarism does exist, the solution to the second problem is critical for the algorithm to be used in practice. We address these two problems in the following two subsections respectively.

4.2 Plagiarism as Subgraph Isomorphism

A detailed examination of the five kinds of disguises in Section 3.1 reveals that while the disguises can significantly alter the source code and the induced token strings, they only insubstantially affect the PDGs. Especially, these disguises will result in a PDG to which the original PDG is subgraph isomorphic. We hence have the following claim for plagiarism detection.

Claim 1. *Restricted to the five kinds of disguises, if g ($g \in \mathcal{G}$) is subgraph isomorphic to g' ($g' \in \mathcal{G}'$), the corresponding procedure of g' is regarded plagiarized from that of g.*

This claim is validated by examining how each of the five kinds of disguises affects the dependence graph. For easy understanding, the PDGs for the original and the plagiarized code in Figure 2 are plotted in Figure 3. Immediately, one can figure out that the left PDG is subgraph isomorphic to the right one. We check how the applied disguises reflect on the PDGs. For clarity in what follows, we use g and g' to denote the PDGs before and after a certain kind of disguises.

- **Format alteration and identifer renaming** do not alter the PDG because neither of them affects the dependencies, so g' is identical to g. For example, although more than twenty format alterations and renamings are applied in Figure 2, the PDG of the original code is preserved.
- **Statement reordering** also leaves the PDG untouched. Two or more statements can be reordered only when they are not bounded by dependencies. Otherwise, reordering could break dependencies, and in consequence cause program errors. As one can see, the statement reordering in Figure 2 is invisible in Figure 3.
- **Control replacement** *generally* leaves g' identical to g. However, when a `while` or a `for` loop is replaced by an infinite loop with a `break` statement, a new program vertex of type "jump" is added to g'. But since the new vertex does not break any existing dependencies, g is still subgraph isomorphic to g'. For example, the two squares in solid lines denote the original `for` and the plagiarized `while` loop. As one can see, the PDG structure is unchanged.
- **Code insertion** introduces new program vertices and/or dependencies into g', depending on what the inserted code does. For program correctness, the inserted code is not supposed to interfere with existing dependencies. Therefore, even though g' can be significantly larger than g (due to code insertion), g is still subgraph isomorphic to g'. For example, the four vertices inside the dotted square in Figure 3(b) correspond to the inserted `for` loop in Figure 2.

Therefore, the five kinds of disguises do not hamper the essential part of the original PDG, although they can significantly alter the code appearance and the induced token strings. Since Figure 3(a) is subgraph isomorphic to Figure 3(b), the code in Figure 2 is regarded as plagiarism according to Claim 1. In fact, the subgraph isomorphism just uncovers what disguises are applied, for example, the variable `count` is renamed as `num`, and the `for` loop is replaced by a `while` loop, etc. In comparison, we feed the two segments of code into both Moss and JPlag, and neither of them recognizes the similarity. This indicates that PDG-based analysis can detect tricky plagiarism that confuses the state-of-the-art tools.

In this way, restricted to the five kinds of disguises, plagiarism detection can be accomplished through checking subgraph isomorphism. However, requiring full subgraph isomorphism may restrict the detection power in practice because trickier disguises beyond the five kinds do exist, and some of them, even though very trivial, can easily confuse

PDG-based detection when full subgraph isomorphism is required. For example, suppose there are two integers, `i` and `j`, which serve as the iteration index in two independent loops, then one can be removed and replaced by the other. Reflected on the PDG, the variable removal merges two vertices, and in consequence, the original PDG is no longer subgraph isomorphic to the plagiarized one. Therefore, for robustness to unseen and unanticipated attacks, we relax CLAIM 1 into the following.

CLAIM 2. *If g ($g \in \mathcal{G}$) is γ-isomorphic ($0 < \gamma \leq 1$) to g' ($g' \in \mathcal{G}'$), the corresponding procedure of g' is regarded plagiarized from that of g', where γ is the* mature rate *for plagiarism detection.*

The mature rate γ is set based on one's belief in what proportion of a PDG will stay untouched in plagiarism. We set it 0.9 in experiments because overhauling (without errors) 10% of a PDG of reasonable size is almost equivalent to rewriting the code. Nevertheless, one needs to understand the code before breaking dependencies.

4.3 Pruning Plagiarism Search Space

In order to find plagiarized PDG pairs, $n * m$ pair-wise (relaxed) subgraph isomorphism testings are needed in principle. However, since most pairs can be excluded through a rough examination, the following two subsections discuss how the search space can be pruned.

4.3.1 Lossless Filter

First, PDGs smaller than an interesting size K are excluded from both $\mathcal{G}$ and $\mathcal{G}'$. For plagiarism detection, we only need to locate PDG pairs of non-trivial sizes, which, if found, can provide enough evidence for proving plagiarism. Second, based on the definition of γ-isomorphism, a PDG pair (g, g'), $g \in \mathcal{G}$ and $g' \in \mathcal{G}'$, can be excluded if $|g'| < \gamma|g|$. These two forms of pruning are lossless in the sense that no PDG pairs worthy of isomorphism testing are falsely excluded.

4.3.2 Lossy Filter

Even through the above two-stage pruning, for any $g \in \mathcal{G}$, there are still multiple $g' \in \mathcal{G}'$, to which g should be checked for (relaxed) subgraph isomorphism. However, since matched PDGs tend to look similar, pairs of dissimilar PDGs can be excluded. This filter is lossy in that some interesting PDG pairs may be falsely excluded.

The similarity measurement must be light-weighted for it to be cost-effective; otherwise, direct isomorphism testing may be more efficient. We therefore take the vertex histogram as a summarized representation of each PDG. Specifically, the PDG g is represented by $h(g) = (n_1, n_2, \cdots, n_k)$, where n_i is the frequency of the ith kind of vertices, and the PDG $g' \in \mathcal{G}'$ is similarly presented by $h(g') = (m_1, m_2, \cdots, m_k)$. We measure the similarity between g and g' in terms of their vertex histograms. A hypothesis-testing based approach is developed in the following, whose advantages over conventional distance-based measurement are discussed in Section 4.3.3.

The major idea is that we first estimate a k-dimensional multinomial distribution $P_g(\theta_1, \theta_2, \cdots, \theta_k)$ from $h(g)$, and then consider whether $h(g')$ is likely to be an observation from P_g. If it is, (g, g') should be checked; otherwise, it is excluded. This judgement is based on a log likelihood ratio test, as outlined below.

The multinomial distribution $P_g(\theta)$ for g is estimated with

$$\theta_i = (1 - \beta)\frac{n_i}{n} + \beta\frac{1}{k} \quad (i = 1, 2, \cdots, k), \tag{1}$$

where $n = \sum_{i=1}^{k} n_i$, and β is a smoothing parameter, and is commonly set as 0.05. We now examine how likely $h(g')$ is an observation from $P_g(\theta)$. For clarity in what follows, we use $X = (m_1, m_2, \cdots, m_k)$ to denote $h(g')$.

We formulate a hypothesis testing problem:

$$\mathcal{H}_0 : X \sim P_g(\theta) \text{ vs. } \mathcal{H}_1 : X \nsim P_g(\theta). \tag{2}$$

Then the generalized likelihood ratio with the observation X is

$$\lambda(X) = \frac{\sup_{\theta \in \mathcal{H}_1 \cup \mathcal{H}_0} L(\theta|X)}{\sup_{\theta \in \mathcal{H}_0} L(\theta|X)}. \tag{3}$$

Because under $\mathcal{H}_0$, the model for X is fixed, then

$$\sup_{\theta \in \mathcal{H}_0} L(\theta|X) = \frac{m!}{\prod_{i=1}^{k} m_i!} \prod_{i=1}^{k} \theta_i^{m_i}.$$

The nominator, on the other hand, achieves its maximum when θ assumes the maximum likelihood estimate, namely,

$$\theta_i = \hat{\theta}_i = \frac{m_i}{m}, \quad m = \sum_{i=1}^{k} m_i. \tag{4}$$

Therefore, the likelihood ratio is

$$\lambda(X) = \prod_{i=1}^{k} (\frac{\hat{\theta}_i}{\theta_i})^{m_i}.$$

Let $T(X) = 2\log(\lambda(X))$,

$$T(X) = 2\sum_{i=1}^{k} m_i \log \frac{m_i}{m\theta_i} \sim \chi^2_{k-1}, \tag{5}$$

where $T(X)$ is the test statistic for G-Test, and asymptotically conforms to χ^2_{k-1} under $\mathcal{H}_0$. A rigid derivation for the asymptotical approximation can be found in [19].

Given a significance level α, we then check whether $T(X) > \chi^2_{k-1}(\alpha)$, where $\chi^2_{k-1}(\alpha)$ is the upper $\alpha * 100$ percentile of the χ^2_{k-1} distribution. If it is, $\mathcal{H}_0$ is rejected, which means that g' is dissimilar to g *under the significance level* α. In consequence, the PDG pair (g, g') is excluded from isomorphism testing. The underlying rationale is that (g, g') by default should be checked unless sufficient evidence is collected against their similarity, and hence against their chance to be an isomorphic pair. In this sense, this hypothesis testing-based filter is conservative, having a low false negative rate.

In fact, the false negative rate can be estimated in the framework of hypothesis testing. According to the interpretation of the Type I error, $\alpha * 100\%$ of all PDG pairs worthy of isomorphism checking are falsely excluded *on average*. For plagiarism detection, false negatives are not a serious problem because as long as some (but not necessarily all) nontrivial plagiarism pairs are found, it is sufficient to support convictions of plagiarism. On the other hand, people who are not comfortable with false negatives can nevertheless lower down the significance level α. When $\alpha = 0$, there are no false negatives, but no further pruning either. The value of the lossy filter is its ability to prune spurious PDG pairs that would otherwise waste much time in isomorphism checking. Therefore, this is a tradeoff between efficiency

and false negatives, and efficiency is usually preferred for plagiarism detection.

4.3.3 Alternatives

There are other alternatives to the above G-test based filtering. For example, one can adopt a distance-based approach: a PDG pair (g, g') is excluded if the distance between $h(g)$ and $h(g')$ is larger than a preset threshold ξ. But in general, no guidance is available for a proper setting of ξ. Moreover, a proper setting of ξ for a particular pair of programs does not generalize to other programs. In comparison, for the hypothesis testing-based filter, the only parameter α is application independent, and can be set in a meaningful way: it balances the pruning power and the false negative rate.

Secondly, people may wonder why the popular Pearson's χ^2 test [13] is not used, given hypothesis testing-based pruning is better than distance-based ones. The reason is that Pearson's χ^2 test is appropriate only when no frequencies in the vertex histogram is near zero because its validity relies on an approximation from a multinomial to a multivariate normal distribution. In our case, since certain m_i's can be or close to 0, Pearson's χ^2 test is thus inappropriate. In comparison, G-test is much more robust than Pearson's χ^2 test [13], and is hence chosen here.

Finally, people may want to fingerprint PDGs with features other than vertex frequencies. We choose vertex histogram just because it is cheap to collect, and performs well. People can index PDGs with structural features, like paths or frequent subgraphs, but this would be computationally expensive, and hence not cost-effective.

4.4 Computational Feasibility

Because our PDG-based plagiarism detection involves subgraph isomorphism testing, we discuss the computation feasibility in this subsection.

Although subgraph isomorphism is NP-complete in general [5], research in the past three decades has shown that some algorithms are reasonably fast on average and become computationally intractable only in a few cases [6] [7]. For example, algorithms based on backtracking and look-ahead, *e.g.*, Ullmann's algorithm [20] and VF [8] are comfortable with graphs of hundreds or thousands of vertices.

Besides the general tractability, the peculiarity of PDGs and the needs for plagiarism detection also lower down the computation workload. In the first place, PDGs cannot be arbitrarily large as procedures are designed to be of reasonable size for developers to manage. Secondly, PDGs are not general graphs, and their peculiarity, like varieties of vertex types, and incompatibility between different types, makes backtrack-based isomorphism algorithm efficient. Lastly, but not the least, for plagiarism detection, the first isomorphism between g and g' suffices, while the conventional isomorphism testing finds *all* isomorphism functions. These three factors make the isomorphism testing on PDGs tractable, and efficient in practice.

Finally, the lossless and lossy filters can effectively toss away spurious PDG pairs from detailed isomorphism testing. In consequence, only a small portion of PDG pairs are really checked. Therefore, our PDG-based plagiarism detection is computationally efficient, although it appears formidable at the first glance.

5. IMPLEMENTATION OF GPLAG

Algorithm 1 GPLAG($\mathcal{P}$, $\mathcal{P}'$, K, γ, α)

Input: $\mathcal{P}$: The original program
$\mathcal{P}'$: A plagiarism suspect
K: Minimum size of nontrivial PDGs, default 10
γ: Mature rate in isomorphism testing, default 0.9
α: Significance level in lossy filter, default 0.05
Output: $\mathcal{F}$: PDG pairs regarded to involve plagiarism

1: $\mathcal{G}$ = The set of PDGs from $\mathcal{P}$
2: $\mathcal{G}'$ = The set of PDGs from $\mathcal{P}'$
3: $\mathcal{G}_K = \{g | g \in \mathcal{G} \text{ and } |g| > K\}$
4: $\mathcal{G}'_K = \{g' | g' \in \mathcal{G} \text{ and } |g'| > K\}$
5: **for each** $g \in \mathcal{G}_K$
6: **let** $\mathcal{G}'_{K,g} = \{g' | g' \in \mathcal{G}'_K, |g'| \geq \gamma|g|, \ (g, g') \text{ passes filter}\}$
7: **for each** $g' \in \mathcal{G}'_{K,g}$
8: **if** g is γ-isomorphic to g'
9: $\mathcal{F} = \mathcal{F} \cup (g, g')$
10: **return** $\mathcal{F}$;

Algorithm 1 outlines the work-flow of GPLAG, a PDG-based plagiarism detection tool. It takes as input an original program $\mathcal{P}$ and a plagiarism suspect $\mathcal{P}'$, and outputs a set of PDG pairs that are regarded as involving plagiarism. In the end, people need to examine these returned PDG pairs, confirming plagiarism and/or eliminating false positives.

At lines 1 and 2, PDGs of the two programs are collected. We wrote a Scheme program to derive and simplify the PDGs from CodeSurfer[1] via its provided APIs. Especially, control dependencies are excluded from consideration in the current implementation for efficiency concerns. Then at lines 3 and 4, PDGs smaller than K are excluded. Finally, from lines 5 to 10, GPLAG searches for plagiarism PDG pairs. For each g that belongs to the original program, line 6 obtains all g''s that survive both the lossless and the lossy filters. And line 8 performs the γ-isomorphism testing. The process from lines 5 to 10 was implemented in C++ based on the VFLib[2]. By default, $K = 10$, $\gamma = 0.9$, and $\alpha = 0.05$ unless otherwise stated. The time reported in the following experiments is the time cost from lines 5 to 10, and is in seconds.

6. EXPERIMENT EVALUATION

In this section, we evaluate the effectiveness and efficiency of GPLAG through experiments. Section 6.1 describes the experiment design and setup, and the following subsections discuss the experimental results in detail.

6.1 Experiment Design and Setup

Subjects	LOC	$\lvert\mathcal{G}\rvert$	$\lvert\mathcal{G}_{10}\rvert$	Description
join	667	17	6	text join tool
bc	8,526	135	56	calculator
less	15,737	386	107	text viewer
tar	18,166	244	83	archive tool

Table 3: Characteristics of Subject Programs

We chose four subject programs for experiments, whose characteristics are listed in Table 3. The number of lines of

[1]http://www.grammatech.com/
[2]http://amalfi.dis.unina.it/graph/

code (LOC) is measured with the tool **sloccount**[3], which excludes both blanks and comments. The third and fourth columns list how many procedures each program has, and how many of them are left with $K = 10$. Finally, the fifth column concisely describes the subject programs.

The join program is mainly used for effectiveness evaluation. We spent two hours plagiarizing it such that both MOSS and JPLAG were confused. In comparison, we show that GPLAG successfully detects the plagiarism (details in Section 6.2). Section 6.3 then focuses on efficiency study, where the three large programs, bc, less, and tar are used. It examines the pruning power of the lossless and the lossy filters, and their implications to the ultimate time cost. Finally, in Section 6.4, we simulate six core-part plagiarism cases with the four subject programs, and evaluate GPLAG's performance in detecting core-part plagiarism. All experiments were carried out on a Pentium 4 PC with 1GB physical memory, running Fedora Core 2. The compiler is gcc-3.3.3 with no optimizations.

6.2 Effectiveness of GPLAG

We compare GPLAG with MOSS and JPLAG for effectiveness evaluation. We plagiarize the program join according to the following recipe. Because all the three tools are robust to format alteration and identifier renaming, these two kinds of disguises were skipped.

Plagiarism Recipe:

1. Whenever m (usually 2 to 4) consecutive statements are not bounded by dependencies, reorder them.
2. Replace a `while` loop with an equivalent `for` loop, and vice versa. Occasionally, a `for` loop is replaced by an infinite `while` loop with a `break` statement.
3. Replace `if(a){A}` with `if(!(!a)){A}`, and `if(a){A}else{B}` with `if(!a){B}else{A}`; recurse if nested `if` block is encountered. Finally, apply DeMorgan's Rule if the boolean expression `a` is complex.
4. Run both MOSS and JPLAG. For any places that they are not confused, insert a statement or a label. Because inserted code breaks the recognized token sequence, code insertion is always effective in confusing token-based algorithms.
5. Finally, run test scripts, and ensure that correctness is preserved during plagiarism.

Although the above recipe suffices to confuse both MOSS and JPLAG, in order to test GPLAG's robustness to tricky attacks, we tried to eliminate redundant code, but finally failed to find any redundant code. In general, we expect that few redundancies exist in mature programs, like join.

The above plagiarism took us about two hours, which suggested that nontrivial work is needed to confuse token-based detections *manually*. However, we notice that the above plagiarism is mechanical to apply, and with some efforts, the plagiarism can be (at least partially) automated. In consequence, confusing token-based algorithms is not laborious any more. This possibility underlines the need for new detection tools that are more robust than token-based ones. Finally, we note that although it sounds irrational to spend two hours plagiarizing a program of 667 LOC, writing a similar program as mature as join will take even much longer.

[3]http://www.dwheeler.com/sloccount/

Procedure	Plagiarism Operations		
	Stmt Reorder	Ctrl Replace	Code Insert
xfields	4	9	0
keycmp	11	7	0
prjoin	12	10	0
join	10	19	2
add_field_list	5	3	0
make_blank	7	1	0

Table 4: Detected Plagiarism Procedures

There are totally 17 procedures in the program. The procedures `main` and `usage` are excluded because interface procedures like them are always ripped off and rewritten in "professional" plagiarism. Among the rest 15 procedures, 9 procedures are filtered out due to their small sizes. Finally, six procedures are left in both the original and the plagiarized versions. Table 4 lists the six procedures, together with what disguises are applied to each of them.

Although these disguises succeeded in confusing both MOSS and JPLAG, the plagiarism was detected by GPLAG in less than 0.1 second. Specifically, it finds six isomorphic pairs, each of which corresponds to one plagiarized procedure. This indicates that plagiarism that slips over token-based checking can be easily detected by PDG-based algorithm, which reaffirms the comparison in Table 2.

6.3 Efficiency of GPLAG

In this subsection, we evaluate the efficiency of GPLAG with the three large programs, bc, less, and tar. Specifically, we take an exact copy of the original program as a plagiarized version. Because PDGs are insensitive to identifier renaming, statement reordering and control replacement, an exact copy is equivalent to a program intensively plagiarized with the aforementioned three kinds of disguises, as far as GPLAG is concerned. We first examine the pruning power of the lossless and lossy filters, and then study the implication of pruning to the ultimate time cost.

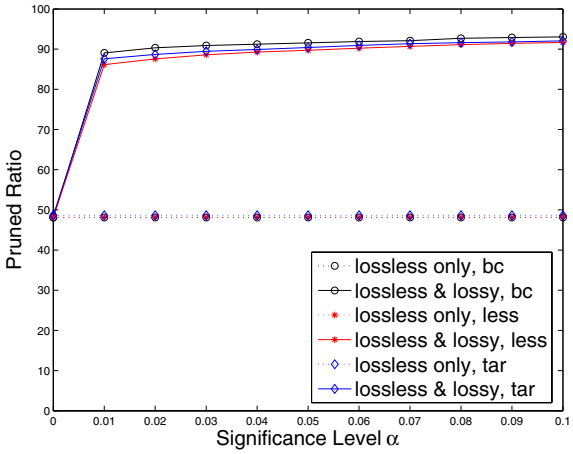


Figure 4: Pruning Ratio of Filters

Figure 4 plots the pruning effect of the lossless and the lossy filters for different programs, when α varies from 0 to 0.1. The y-axis is the pruned ratio, *i.e.*, what percentage of PDG pairs are excluded from $\mathcal{G}_K \times \mathcal{G}'_K$. The horizontal dotted lines show the pruning effect of the lossless filter on

Subjects	No Filter			Lossless Only			Lossless & Lossy		
	Tested Pairs	Time	Matches	Tested Pairs	Time	Matches	Tested Pairs	Time	Matches
bc	3,136	251.21	63	1724	251.78	63	293	0.056	60
less	11,449	1171.35	125	6,304	1170.3	125	1,288	7.38	114
tar	6889	853.1	110	3759	850.24	110	722	122.61	89

Table 5: Efficiency of GPLAG

the three programs. Roughly, about one half PDG pairs are pruned with the lossless filter only. The solid lines plot the pruning effects when the lossy filter is also employed. The curve rocketing from $\alpha = 0$ to $\alpha = 0.01$ indicates that a great proportion of PDG pairs that survive the lossless filter are in fact quite dissimilar, and do not need isomorphism testing. When α gets larger, more PDG pairs are pruned, but at a mild rate.

We now examine how the pruning affects the ultimate time cost. Table 5 lists the efficiency comparison on the three subject programs when 1) no filter, 2) only lossless, and 3) both lossless and lossy filters are applied. For each of the three kinds of filter application, we record how many PDG pairs are actually tested, the time cost, and the number of found matches. The time is *capped* in the sense that a timeout of 100 seconds is set for every isomorphism testing. Timeout is necessary because subgraph isomorphism testing can still take hours (or even longer) to finish for some cases. A PDG pair whose isomorphism testing fails to terminate within 100 seconds is called a *time hog*, and is regarded unmatched.

We now examine Table 5 in detail. The first interesting point it suggests is that the lossless filter alone does *not* save much time, although it cuts off about one half of the PDG pairs (Figure 4). Specifically, the saving is only a few seconds. The explanation is that for PDG pairs pruned by the lossless filter, the isomorphism testing algorithm also recognizes the impossibility of isomorphism within a few steps of search. On the other hand, when the lossy filter is applied, the time cost significantly shrinks. With a time cap set as 100 seconds, and most isomorphism testings finish within tens or hundreds of milliseconds, we know that the time saving mainly comes from the avoidance of time hogs, rather than from the pure reduction of PDG pairs. Therefore, the lossy filter also helps circumvent time hogs while tossing away spurious PDG pairs.

6.4 Core-Part Plagiarism

Carriers		Lossless Only		Lossless & Lossy	
Orig.	Plag.	Time	Matches	Time	Matches
bc	less	245.457	10	2.78	8
bc	tar	620.394	19	0.267	8
less	bc	1069.72	30	224.48	9
less	tar	2655.13	45	137.83	7
tar	bc	1084.34	14	0.621	9
tar	less	1048.43	11	204.16	7

Table 6: Simulated Core-Plagiarism

In this section, we simulate cases of core-part plagiarism with the four subject programs, and examine the effectiveness and efficiency of GPLAG in detecting core-part plagiarism. Specifically, we treat the three large programs as carrier programs, and embed the original and the plagiarized versions of the six procedures (listed in Table 4) into two different carrier programs. GPLAG is expected to find the six plagiarized procedures both accurately and efficiently.

Table 6 presents the experiment results for the six carrier combinations, with and without the lossy filter. The six matches are all detected in the six carrier program combinations. Clearly, with the lossy filter, much fewer false positives are alarmed. As to the time cost, similar to the result in Table 5, the lossy filter significantly reduces the time cost for these simulated core-part plagiarisms. Again, this time reduction is due to the avoidance of most time hogs. Therefore, the lossy filter is critical for GPLAG to detect core-part plagiarism in large programs: Not only can it reduce the false positive rate, but it also makes time cost acceptable.

7. DISCUSSIONS

In this section, we discuss the related work and potential industry implications of our work.

7.1 Related Work

This study is closely related to the previous work on plagiarism detection. In Section 3.2, we provided an overview of existing techniques based on different representations of programs [1, 2, 9, 11, 17, 18]. We show that GPLAG is more effective than these methods, both conceptually and experimentally. The program dependence graph, first proposed by Ferrante et al. [4], has previously been used in the identification of duplicated code for the purpose of software maintenance [10, 12]. In this study, we propose GPLAG as a PDG-based algorithm for plagiarism detection. Moreover, for both effectiveness and efficiency, a statistical lossy filter is developed, which has not been seen in previous studies [10, 12]. However, on the other hand, since GPLAG involves graph analysis, it is nevertheless less efficient than those based on sequence analysis. But this difference has only little impact in practice, because GPLAG usually terminates within seconds even when subject programs are of thousands of lines of code. There are studies on detection of other kinds of plagiarism, such as plagiarized research papers, homework answers, and Web pages [3]. The nature of such plagiarism is rather different from that of software plagiarism which often requires more sophisticated analysis.

From the data analysis point of view, GPLAG is related to graph mining. Graphs have been adopted for data modeling in many domains, and many graph mining algorithms are developed. However, these algorithms cannot be used for plagiarism detection because they search for potential candidates almost everywhere in a "candidate-generation-and-check" approach. As an example, given two identical PDGs with 30 vertices and 43 edges, CloseGraph [21] fails to terminate *in two days*. This paper proposes an isomorphism-based approach, which proves both effective and efficient.

Finally, this study falls into an emerging application do-

main in data mining: data mining for software engineering. Previous research indicates that proper mining of software data can produce useful results for software engineers. Livshits et al. apply frequent itemset mining algorithms to software revision history, which uncovers programming rules that developers are expected to conform to [16]. Liu et al. show that mining program control flow graphs can help developers find logic errors [14,15]. These cases well exemplify the promise and usefulness of data mining in software engineering. This study provides yet another such example.

7.2 Implications to Software Industry

Software plagiarism has been an important issue in software industry for intellectual property and software license protection, especially for open source projects. Thus it is important to develop robust and effective approaches to software plagiarism detection. Our study shows that previous approaches that rely on string matching, parse-tree construction, and tokenization cannot handle sophisticated control flow alternation and code insertion, and thus cannot be effective at fighting against "professional" plagiarists.

This study proposes a rather different approach, GPLAG, which bases the analysis on program dependence graphs. The study demonstrates its effectiveness at the detection of sophisticated plagiarism in comparison with the current state-of-the-art tools. Moreover, our experiments show that this approach is also scalable to large programs. Even in practice when software of millions of lines of code is encountered, GPLAG may be still applicable because one component only needs to be compared with its counterpart. Finally, GPLAG can be easily extended to other programming languages. What one needs for a new language is merely a parsing frontend, from which PDGs can be derived. Currently, the frontends for C, C++ and Java are available and ready to use. To this extent, GPLAG is not only a program for idea demonstration but also a practical tool.

One may wonder whether GPLAG can be really used for lawsuit conviction. In the experiments, we have witnessed low false positive rate. In general, chances are slim that one PDG is isomorphic to another by chance. Even for the same task, two developers will likely come up with different implementations, and different PDGs in consequence. Therefore, if GPLAG judges one case as plagiarism, the chance could be high. However, human participation is nevertheless needed for result verification and final judgement.

8. CONCLUSIONS

This paper proposes a new plagiarism detection algorithm, GPLAG, which detects program plagiarism based on the analysis of program dependence graphs. Experiments have well demonstrated its effectiveness over existing tools, and its applicability in practice.

9. REFERENCES

[1] B. S. Baker. On finding duplication and near duplication in large software systems. In *Proc. of 2nd Working Conf. on Reverse Engineering*, 1995.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of Int. Conf. on Software Maintenance*, 1998.

[3] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang. Finding interesting associations without support pruning. In *Proc. of 2000 Int. Conf. Data Engineering*, 2000.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[5] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[6] C. Hoffman. *Group-theoretic Algorithms and Graph Isomorphism*. Springer Verlag, 1982.

[7] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proc. of 6th ACM Symp. on Theory of Computing*, 1974.

[8] J. E. Hopcroft and J. K. Wong. Performance evaluation of the VF graph matching algorithm. In *Proc. of 10th Int. Conf. on Image Analysis and Processing*, 1999.

[9] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7), 2002.

[10] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. of 8th Int. Symp. on Static Analysis*, pages 40–56. Springer-Verlag, 2001.

[11] K. Kontogiannis, M. Galler, and R. DeMori. Detecting code similarity using patterns. In *Working Notes of 3rd Workshop on AI and Software Engineering*, 1995.

[12] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. of 8th Working Conf. on Reverse Engineering*, 2001.

[13] E. Lehmann. *Testing Statistical Hypotheses*. Springer Verlag, 2nd edition, 1997.

[14] C. Liu, X. Yan, and J. Han. Mining control flow abnormality for logic error isolation. In *In Proc. 2006 SIAM Int. Conf. on Data Mining*, 2006.

[15] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *Proc. 2005 SIAM Int. Conf. on Data Mining*, 2005.

[16] V. B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proc. of 13th Int. Symp. on the Foundations of Software Engineering*, 2005.

[17] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *J. of Universal Computer Science*, 8(11), 2002.

[18] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 2003.

[19] R. R. Sokal and F. J. Rohlf. *Biometry: the principles and practice of statistics in biological research*. Freeman, 3rd edition, 1994.

[20] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. of the Association for Computing Machinery*, 23(1), 1976.

[21] X. Yan and J. Han. CloseGraph: mining closed frequent graph patterns. In *Proc. of 9th Int. ACM SIGKDD Conf. on Knowledge Discovery and Data Mining*, 2003.