Acta Cybernetica **19** (2009) 191–216.

Plagiarism Detection in Source Programs Using Structural Similarities

Gergely Lukácsy* and Péter Szeredi*

Abstract

The paper presents a plagiarism detection framework the goal of which is to determine whether two programs are similar to each other, and if so, to what extent.

The issue of plagiarism detection has been considered earlier for written material, such as student essays. For these, text-based algorithms have been published. We argue that in case of program code comparison, structure based techniques may be much more suitable. The main idea is to transform the source code into mathematical objects, use appropriate reduction and comparison methods on these, and interpret the results appropriately.

We have designed a generic program structure comparison framework and implemented it for the Prolog and SML programming languages. We have been using the implementation at BUTE to successfully detect plagiarism in homework assignments for years.

Keywords: plagiarism, program source, graph similarity

1 Introduction and motivation

Comparison of essays and other written materials has been in focus in recent years [27]. Detecting plagiarism in written materials is an issue in education as well as in law procedures. World wide public polls show that two-thirds of university students have used other people's ideas in an impermissible way at least once during their studies. Law disputes include the SCO-IBM debate over the allegedly unauthorised use of portions of the AIX operating system in Linux.

Regrettably, several sites on the Internet provide free or low cost, quick and efficient access to written materials of many types. Unbelievably, sites such as `CheatHouse`[1] or `SchoolSucks`[2] proudly provide tons of essays, dissertations, reports, etc. for students looking for an easy way to have their assignment of some sort fulfilled. We do agree that it is a good idea to get acquainted with the area one

*Budapest University of Technology and Economics (BUTE), Department of Computer Science and Information Theory, 1117 Budapest, Magyar tudósok körútja 2., Hungary, Phone: +36 1 463-2585 Fax: +36 1 463-3157, E-mail: {lukacsy,szeredi}@cs.bme.hu

[1] `http://www.cheathouse.com`

[2] `http://www.schoolsucks.com`

DOI: 10.14232/actacyb.19.1.2009.13

is interested in by reading similar materials. However inspiring someone to cheat is a different issue.

In case of programming assignments, it is important to detect the duplication of programs or parts of these. Students attending the course "Declarative Programming" at BUTE are expected to hand in a major programming assignment at the end of the semester. This means mass amount of program sources year by year.

Checking these programs by hand seems to be beyond possibility. Having n programs we should check $\frac{n*(n-1)}{2}$ pairs to have all the cases covered. Notice, that we really should check all of the pairs, because the relation "P1 is similar to P2", where P1 and P2 are programs, is not transitive. This practically means that even if we know that source A is similar to source B and source B to source C we cannot draw any direct conclusion about the similarity degree of sources A and C.

Luckily, in our particular case several assignments can be excluded from the whole set. For example, we do not care whether two bad solutions are similar or not (a solution is bad if it does not solve a certain percentage of the given test cases). However we still have $O(n^2)$ pairs to test manually, where n is often greater than 100.

Our aim was to develop methods and tools to assess the similarity of programs in order to narrow down the need for manual testing to an acceptable amount. We have defined the notion of a *similarity degree* which reflects how much two programs match. For the methods to be generic and flexible enough we have developed a *multi phase comparison framework*.

The actual comparison is performed between mathematical entities where the meaning of similarity can be formally specified. These entities are generated from the programs to be compared. The procedure may vary for different programming languages, so separate front-end modules should be developed for each language. Naturally, the mathematical entities must be generic and powerful enough to be applicable to different languages. We have chosen directed, labelled graphs for this purpose. Now, the comparison of source programs is actually reduced to calculating the similarity measure of graphs. Notice, that this way it is also possible to determine the similarity degree of two programs written in different languages.

The framework is customisable, so that it remains usable under varying circumstances. For instance, in case of shorter programs a different similarity threshold may be more appropriate than in the case of bigger ones. Moreover, we found that applying certain well selected simplifying graph transformations, called *reductions*, has favourable effects on the efficiency of the approach. Such reductions include removing specific nodes and edges and thus creating higher level, more abstract views of the programs.

The structure of the paper is as follows. In Section 2 we give a brief comparison of our approach with other ongoing research work. Next, we describe what we expect from a plagiarism detection framework, i.e. what are the types of student tricks it should be resistant to. In Section 4 we give an overview of the proposed framework and introduce the main concepts. Following this, we describe the three components of the framework: the *Front-end module*, the *Simplifier* and the *Comparator*. Section 5 describes the prototype implementation of the framework for

Prolog and SML programs. Next, we evaluate the system and show execution results. Finally, we give a summary of our work.

2 Related work

Several solutions exist for detecting plagiarism in written documents (like iThenticate [16], FindSame [15], CopyCatch [14], SCAM [26] or the new Hungarian portal from the Computer and Automation Institute of the Hungarian Academy of Sciences called KOPI [5]). However, this is not the case for program sources. A reason for this may be that it is widely believed that detecting plagiarism in programs is much easier than in free text. This is because programming languages are formally defined and, as opposed to the case of free text, it is generally assumed that people use only a few tricks to hide the fact of plagiarism.

Alan Parker and James Hamblen in [23] explicitly say that copied software is "a program which has been produced from another program with a small number of routine transformations". These routine transformations include modifying the comments, changing the names of the variables or (in the worst case) changing the control structures (e.g. using `while` instead of `for`). The suggested technique for comparing programs is the following:

1. Get rid of every comment in the source codes.
2. Get rid of every useless new line, white space, etc.
3. For each pair of source programs use a normal UNIX diff program, which compares the files line by line.
4. Examine the results.

In [8] J. A. Faidhi and S. K. Robinson suggested a scale which defines the level of plagiarism (L0-L6) based on what kind of modifications the cheater used. For example, we obtain L1 from L0 by modifying the comments, L2 from L1 by further modifying the variable names as well, etc. This scale is often used by programs for plagiarism detection to "position" themselves.

Most existing software solutions are based on statistical or lexicographic approach where, for example, they compare identifiers with identifiers to determine how similar the source programs are. Such systems are the DUP [2], SIM [9], SIFF [3] or Bandit [28].

On the other hand, approaches based on structural properties were already proposed several decades ago. For example, in [4] J. M. Bieman and N. C. Debnath suggested building program graphs, while T. J. McCabe proposed [20] to compute a characteristic numeric value, a metric, for each program code according to its complexity (which was based on the number of computation paths available within the program). This metric is widely known today as *cyclomatic complexity*.

Further programs that support structure comparison include the *Plague* [29], the *YAP* (Yet Another Plague) series [30], and the *Moss* (Measure Of Software Similarity) [25] program. Plague builds so called structure profiles for source codes

and compares them. The YAP programs implement a two phase approach. First they convert the source programs into a more unified form, e.g. removing comments, translating upper-case letters to lower case. In the second phase (depending on the actual YAP version) they apply algorithms, such as Heckel's isolation technique [12], that are resistant to specific structural changes, e.g. changing the order of independent statements or replacing a procedure call by the procedure body. The authors of Moss have developed a general algorithm for calculating a so called fingerprint from an arbitrary document which they claim to be especially precise in case of source programs.

Paper [21] introduces an XML-based model called XPDec (XML Plagiarism Detection Model) suitable for programs written in a procedural language such as C or Pascal. XPDec uses XML to represent structural properties of the source programs and is useful for detecting common forms of reordering plagiarism. An extended version of this approach is presented in [22] which takes also into account the structure of the control sequences in the source programs.

Plagiarism detection is also closely related to code duplication detection. Here, the idea is to detect when developers use previously existing code which solved a problem similar to the one they are currently trying to solve. This may indicate a design problem as the duplicated code is difficult to maintain (e.g. fixing bugs must be done in several places). Although most of the existing solutions for duplication detection are based on the lexicographic approach, some of them use the structural properties of the source codes [24, 18].

Our approach introduced in this paper belongs to the group of program plagiarism systems utilising the structural properties of the source programs. However, instead of providing sophisticated comparison techniques that are resistant to the most common tricks we apply so called reduction steps to create more abstract views of the programs. These views are then compared by using relatively simple comparison algorithms. We argue that this approach makes our system fairly efficient and easy to customise.

3 Goals

We now discuss the most significant student tricks we believe a plagiarism detection framework should be resistant to. To illustrate such tricks in a language independent way we use pseudo-language examples below. We realise that there exist tricks only applicable for specific programming languages. Handling these is the task of the concrete implementation of such a framework (cf. Section 4).

Changing the names of identifiers and variables is the most common trick. A piece of source code which contains only single letter variable names may look rather confusing and tangled. However, it can be easily transformed into a program which uses talkative names. For humans, sometimes only this is enough to hide the fact of plagiarism. A similar trick is to change the natural language in which the program identifiers are formulated: use English names in one program and use another language in the other. It is also possible to change not just the variable, but the

function and/or predicate names, too. For example, it is very easy to transform the function

```
void solve_the_problem(Input_data, Results) {...}
```

... to the following:

```
void do(Input, Output) {...}
```

One can also change the number of arguments (the so called *arity*) of the functions, without affecting the code. For example one can use dummy parameters, which are set to something irrelevant at call time. If one changes not only the name of a function, but also its arity, it may become really difficult for the human to recognise that it is semantically equivalent to some other function.

Sometimes it is profitable for students to cut the code into several pieces and place them into separate files using the module system of the given language. Similarly, reordering the sequence of the function definitions in a source file is an easy, but often effective trick. Students also like to change the order of statements in the body of a function if these statements do not depend on each other: for example, two independent variable assignments can be switched. In case of logic programming languages, this kind of trick is very common as a body of a predicate is the logical conjunction of so called goals. This means that these goals can often be reordered freely without effecting the execution of the program.

Putting useless functions into the code may also be used to disguise plagiarism. For example we can "borrow" some code from another program which has nothing to do with the current programming assignment. Computer based methods may find this disturbing, because this technique introduces new variables and functions, changes the size of the file, etc. Sometimes one can recognise this trick by doing static source code analysis and detecting that these functions are never called, but this is not true in general.

Consider the following example, where the procedure `calculate` will never be called. This procedure can be anything, most likely a piece of some big code, with the only aim to conceal the fact that the original source code for `solve_the_problem` was made by some other individual.

```
int solve_the_problem(A, B) {
    if (A > 0) {
    ...
    X  = A + 35;
    ...
    if (X < 0)                  // X cannot be negative here
      calculate(X, B);
    else
      X = 2;
    ...
    }
    ...
}
```

In the general case those parts of the program which are never called can only be detected at run time. Unfortunately, even if we detect such code fragments it does not mean that we found an instance of plagiarism. Sometimes such code is simply the result of programming errors, which even the author of the program is not aware of.

Analogously to placing useless procedures in the program code one can place useless calls in the body of a procedure without changing its task. In the following example we show two totally useless lines inserted into a function, not changing the execution of the program:

```
...
C = 2; A = 3-C;                    // A = 1
...
if (C == A+1) {                    // check if C = 2
...
```

Finally, we show two tricky, but easily implementable types of program transformation. The first we named *call-tunneling*, while the second *call-grouping*. Call tunneling is based on the idea that instead of letting function `A` to call function `C` directly, we insert an intermediate function `B`. In this new scenario `A` calls `B` and `B` calls `C`. If function `B` returns what it got from `C` without any modification, then the transformed program will be equivalent to the original one. Call-tunneling is very hard to detect, because, for example, function `B` is actually called during the execution, therefore it seems to be an important part of the program.

Call-grouping is a simple technique to significantly modify the structure of a program even if one does not really understand what the code actually does. The main idea is very similar to that of call-tunneling: if there is a function which calls several others, we can regroup these calls into some new functions to produce a totally different code structure. Let us consider the following piece of code:

```
int original_function(A, B) {
  T = call1(A);
  Q = call2(B, T);
  E = call3(Q);
  Z = call4(A, E);
  return call5(Z);
}
```

Using call-grouping one can transform it to the following equivalent program.

```
int groupped_function(A, B) {
  E = temp1(A, B);
  return temp2(A, E);
}
```

```
int temp1(A, B) {
  T = call1(A);
  Q = call2(B, T);
  return call3(Q);
}

int temp2(A, E) {
  Z = call4(A, E);
  return call5(Z);
}
```

Notice that functions `call1`,...,`call5` are invoked in the same way as in `original_function`, but two new grouping functions are also introduced.

4 The framework

The proposed framework consists of three main components which are handled by independent program modules:

1. *Front-end*: performs source code to model *mapping*
2. *Simplifier:* carries out model *reduction*
3. *Comparator:* does model *comparison*

The *Front-end* creates a mathematical entity — which we call a **model** or an **abstract view** — from the source program to be examined. Subsequently, these views can be reduced in many ways by the *Simplifier*, creating different **abstractions** of the original model. Having the abstractions of two source programs, we use the *Comparator* to compare models on the same abstraction level and determine a similarity degree (a number between 0 and 1). As the abstraction becomes higher, the similarity of the abstract views is less and less indicative of the similarity of the original programs. Therefore we assign a factor (again a number between 0 and 1) to each abstraction level, with which we multiply the similarity degree obtained earlier.

Figure 1 shows the overview of the proposed framework. Here we start from two source programs `source A` and `source B`. The *Front-end maps* these sources to two models, `model A` and `model B`. Higher and higher abstractions of these models are produced by *reductions*, using the *Simplifier*. Finally, the models on the same abstraction levels are compared with each other.

In the following subsections we discuss in detail the main parts of the framework.

4.1 Source code to model mapping

In general, the entity to which a program source is mapped can be chosen arbitrarily. For example, let us consider the size of the program source (e.g. in terms of characters used) as an abstract entity characterising the program, and consider the advantages and disadvantages of this choice. It is true that if we

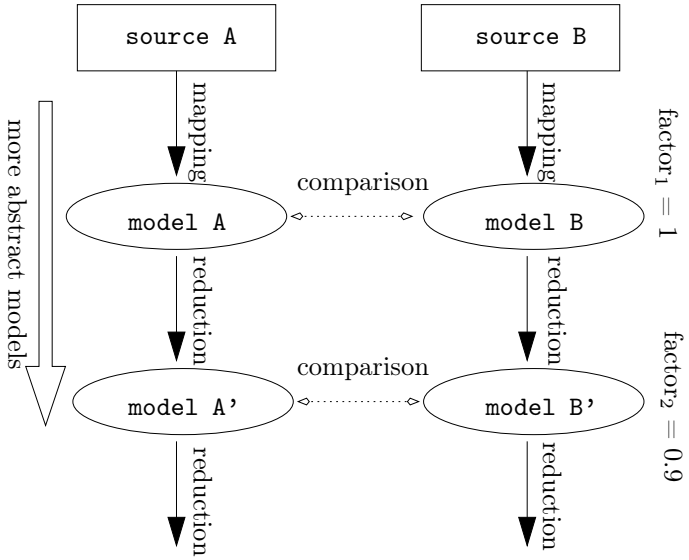


Figure 1: Overview of the proposed framework

examine two entirely identical programs, then the comparison of their abstract views will signal match (the sizes of the programs will be the same). It also sounds feasible to consider the two program instances suspicious, if their size, in terms of characters, is exactly the same. However, if the programs are similar, but not identical, then the program size abstraction cannot give any hint on their similarity.

A further issue is that of simplifying transformations. When a program is characterised by its size, practically no further simplifications can be applied. The only, very weak option is to make further abstractions by rounding the size, e.g. using 1 kbyte instead of 1324 bytes.

Therefore, the abstract view must be more sophisticated (to allow diverse abstraction levels) and, more importantly, it must be possible to draw conclusions on the similarity of the programs from the similarity of the abstract views.

Therefore we suggest the use of *directed, labelled graphs* as the abstract views characterising the programs. Here the meaning of nodes, edges and their labels may vary from implementation to implementation. For example, the abstraction may be the program call graph, the data-flow graph of an execution, or — in case of object-oriented languages — the graph describing the object structure. The labels are used to describe the properties of the nodes and edges, e.g. to express that a node represents a built-in entity and not a user function.

Note that we suggest to ignore the labels in graph comparison, as we would like the similarity measure to focus on the graph structure. A further benefit of this is that it makes the comparison algorithms simpler and faster. However, the reduction steps do use the information stored in labels. This may result in somewhat strange

effects: two graphs, that are considered isomorphic on one abstraction level, become non-isomorphic on the next level, provided the given reduction step uses the labels.

The graph representation is general enough to describe any kind of entity. As an extreme, even our first example, the program size abstraction, can be described as a labelled graph (with a single node whose label is the size).

4.2 Model reduction techniques — abstraction levels

One can envisage some kind of perfect mathematical models, that contain every bit of information present in the program source code. In this case we can be sure that, when two such perfect models are isomorphic, the corresponding program source code is the same. Of course, such a model is nothing else, but the source code itself in a different representation.

For any programming language and for any specific piece of source code, the lowest abstraction level, which we call level 0, could be considered to contain perfect models only. At first, one may think that the best one could do is to directly compare such perfect models. However, this may require a very sophisticated comparison algorithm, which is on one hand fast and easy to customise, and on the other hand resistant to the possible cheating methods mentioned in Section 3. Instead, we decided to follow a different approach using a series of views with increasing abstraction levels.

We thus propose to use several abstraction levels (as shown in Figure 1) and use *relatively simple and fast comparison algorithms* between models on the same level. Higher abstraction levels are built from lower ones (possibly utilising the labels in addition to graph structure) using certain transformations, called *reduction* steps. Our task is to transform the initial perfect models to ones which are more and more resistant to specific tricks, and which still represent the original program sources as much as possible.

Naturally, reduction steps are destructive operations: with every bit of dropped information we widen the gap between the perfect model and the model in question.[3] Because of this, a perfect match (isomorphism for example) between two models on a high abstraction level "means less" than the same type of match on a lower level. To handle this, we assign a factor to each abstraction level in question, with which we multiply the similarity degree achieved on that level.

We define the similarity of two programs as

$$\max_{1 \le i \le n} F_i S_i \tag{1}$$

where n is the number of abstraction levels in the concrete implementation of the framework. F_i is the factor assigned to abstraction level i (a number between 0 and 1) and S_i is the actual similarity degree obtained on abstraction level i (also a number between 0 and 1). We require that $F_{i+1} < F_i$ holds for any i, i.e. the factors

[3] In theory we may end up in a point where every model becomes a singleton graph (a graph consisting of a single node): on this level every pair of models is isomorphic.

1. $i = 1, Max = 0$
2. compare the two models on abstraction level i, i.e. calculate S_i
3. calculate $Max = max(S_i * F_i, Max)$
4. in case of isomorphism ($S_i = 1$), exit with the output value Max
5. if $Max \geq F_{i+1}$, exit with value Max, otherwise $i = i + 1$ and goto step 2

Figure 2: The algorithm for determining the similarity degree of two programs

assigned to the abstraction levels form a strictly monotonic decreasing sequence. However, we do not pose any restrictions on the values S_i and S_{i+1}.

To determine the maximum, we may simply calculate expression (1). For example, let us assume we have two abstraction levels, level 1 and 2 with factors 1 and 0.9 respectively. If our models are assigned a 98% similarity on level 1 and are isomorphic on level 2, the algorithm calculates the values $0.98 * 1 = 0.98$ and $1 * 0.9 = 0.9$ respectively. The final result is the larger of these, namely 0.98.

We can optimise this naive algorithm in the following way. Whenever we detect that the maximum value we may obtain in the next abstraction level (which is F_{i+1} as $S_{i+1} \leq 1$ holds) is less or equal than the current maximum value Max, we can stop. This is because the factors are decreasing, thus for every $j = i + 1, \ldots, n$ it holds that $F_j * S_j \leq Max$. This trivially means that if we detect isomorphism between two models at abstraction level i we can immediately finish execution. When we stop, the final result (i.e. the similarity degree of the source programs in question) is the current maximum Max. This algorithm is shown in Figure 2.

4.3 Model comparison algorithms

In Section 4.1 we argued that directed, labelled graphs are good mathematical constructs for describing models of programs. Considering this, the concrete comparison algorithms are most likely related to graph theoretical algorithms.

In general, our task is to define in what extent are two graphs *similar* to each other. Let us first consider the problem of graph isomorphism as an extreme case of graph similarity.

4.3.1 Graph isomorphism

The problem of graph isomorphism is the following. Given two graphs, G and H, we look for bijection f between the nodes of the graphs, so that (x, y) is an edge in G if and only if $(f(x), f(y))$ is an edge in H.

The graph isomorphism problem belongs to the class of NP problems, but we still do not know if it is NP-complete [1]. However, in special cases we know the complexity exactly or at least we can produce algorithms which run with acceptable

To calculate the code $C(T)$ of tree T:

1. determine the child subtrees of the root of T, N_1, N_2, ..., N_k
2. determine the codes for N_1, N_2, ..., N_k
3. sort the codes $C(N_1)$, $C(N_2)$, ..., $C(N_k)$ into ascending order using their binary values for ordering. Assuming the concatenation of these produces a sequence S, return the sequence $1S0$ as the code assigned to T

Figure 3: The algorithm for calculating the code of a tree.

speed. For example we know polynomial algorithms for planar graphs as well as for graphs where the maximum vertex degree is bounded [7].

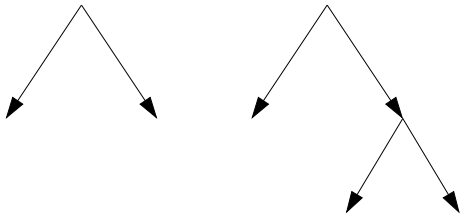
In case of trees a more straightforward approach is applicable [17]. Namely, it is possible to construct a code in *linear time* for two trees T_1 and T_2, which fulfils the following two criteria:

1. if T_1 is isomorphic to T_2, then the code of T_1 equals to the code of T_2
2. if the code of T_1 equals to the code of T_2, then T_1 is isomorphic to T_2

Actually creating a tree code is nothing more than applying a geometrical transformation that maps a 2D tree to a one dimensional sequence of two characters. One can use the digits "0" and "1" or the parentheses "(" and ")" as the elements of the sequence, and accordingly the code of a leaf is "10" or, when parentheses are used, "()". Let T be the tree to be encoded and let $C(T)$ denote the code assigned to the tree T. Now, the recursive algorithm presented in Figure 3 assigns a binary number to any tree T.
Two examples of such encoding are given in Figure 4. We note that sometimes it is useful to apply a special notation for the leaves, to distinguish these from the code corresponding to other parts of the tree. We will use letter `L` for this purpose. Accordingly, the codes in Figure 4 can be written as `1LL0` and `1L1LL00`, respectively.

It is important that the algorithm in Figure 4 can also be used for DAGs (Directed Acyclic Graphs), i.e. directed graphs, not containing directed circles. In this case, in addition to a DAG, the algorithm requires that a "root node" is specified, which serves as a starting point for the algorithm. An example of such a graph (the starting node is denoted by R) and its code can be seen in Figure 5.

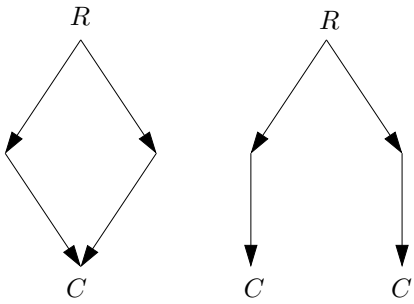
Note that the code of a DAG can also be obtained by first transforming the DAG into a tree, and then taking the code of this tree. The transformation takes a vertex a with $n > 1$ incoming edges and replaces it by n new vertices, each with a single incoming edge (and each new vertex inherits all the outgoing edges of the original one). By repeating this transformation step we can eliminate all vertices with multiple incoming edges and thus obtain a tree from the DAG. The right hand



Code: `110100` Code: `1101101000`

Figure 4: Two examples for the coding scheme

side of Figure 5 shows the result of this transformation process, when applied to the DAG on the left hand side.



Code: `11L01L00`

Figure 5: A DAG and the corresponding tree with its code

4.3.2 Graph similarity

Checking isomorphism is usually not enough by itself. The reason is that we cannot expect (at least on the lower abstraction levels) that the program graphs will be totally isomorphic, even with the most sophisticated source code to model mappings and reduction techniques. Actually we would like to detect if two graphs of hundreds of nodes (which are very typical for the programs we use) at a given abstraction level are *nearly* identical.

The general approach we use is to check how it is possible to transform one DAG code to another. For example, let us consider the following two sequences that correspond to the codes in Figure 4.

First sequence (F): `1L1LL0`

Second sequence (S): `1LL0`

The transformation steps we need to e.g. convert the first sequence into the second one can be described as follows: "remove `1` and then `L` from position 3 and 4 in sequence F". Such transformation steps can be constructed by e.g. using algorithms solving the so called *longest common subsequence (LCS)* problem [13]. Let us denote by $\Delta(A, B)$ the transformation steps between two arbitrary sequences A and B. $\Delta(A, B)$ is a set, containing pairs. The first part of such a pair can be `0`, `1` or `L`, the element to be added or removed. Each of these elements is preceded by either a plus(`+`) or a minus(`-`) sign, corresponding to element addition and removal. The second part of a pair is an integer, describing the position the specific transformation step should be applied at. In our case, $\Delta(F, S)$ is the set $\{(\texttt{-1}, 3), (\texttt{-L}, 4)\}$.

To determine the similarity degree of two arbitrary codes we assign penalties to the specific transformation steps. For example, we may say that the removal of a leaf (i.e. a `-L` in the transformation set) reduces the similarity by a certain amount, let us say by 0.01. Using the penalties we calculate $\Omega(A, B)$, the discrepancy function describing to what extent codes A and B are different:

$$\Omega(A, B) = min\left(1, \sum_{(E,\ldots)\in\Delta(A,B)} P(E)\right) \tag{2}$$

Here P is the penalty function that assigns a value to a given type of transformation step. Using $\Omega(A, B)$ we define the *similarity degree* of graphs A and B as:

$$1 - \Omega(A, B) \tag{3}$$

Let us note that what we actually calculate here is a variant of the so called Levenshtein and edit distances. The Levenshtein distance [19] between two strings is the minimal number of operations needed to transform one string into the other. By operation we mean an insertion, deletion, or substitution of a single character. The edit distance [6] is a generalisation of the Levenshtein distance in that the operations have costs assigned to them, similar to the costs we have defined above.

4.3.3 Distinct paths in the graphs

Unfortunately, in a special case very similar graphs are considered to be far from each other according to the similarity degree introduced above. The reason for this is the way how DAG codes are built. We have seen in Figure 5 that the code corresponding to node C appears in the code of the whole DAG twice (i.e. we have two `L` characters in the code, although the original DAG has only one leaf). In general, for any DAG G, the code of a node v appears in the code of G exactly m times, where m equals to the number of distinct paths from the root to v.

Let us assume that the DAGs corresponding to programs A and B differ only slightly in a single node, which is, however, accessible from the root along many distinct paths. Because of the reasons outlined above, the DAG codes corresponding to A and B will differ significantly.

We suggest two ways to overcome this shortcoming, both of which are used in our prototype system described in the next section. First, we suggest to use

reduction steps which decrease the number of distinct paths from the root, thus making the graphs more "tree like". For example, filtering multiple edges in a graph (cf. Section 5.2) reduces the number of distinct paths significantly.

As a second solution we suggest to introduce a slight modification of the similarity degree as defined in (3). This modification relies on identifying nodes with numerous incoming edges (let us call these nodes *popular*)[4]. Using structural decomposition we suggest to calculate a variant of the discrepancy function Ω specified in Equation 2, called Ω'. The final similarity degree will thus be $1 - \Omega'(A, B)$.

We now describe how to compute the value of $\Omega'(A, B)$ for arbitrary two graphs A and B. We assume that a popular node N in A can be associated with its counterpart M in B. In our implementation we use a very simple heuristics for this: we pair those popular nodes whose number of incoming edges and number of arguments are the closest (proving the mathematical properties of this heuristic is a future work). If such a pairing is not possible (if at most one of the graphs contains popular nodes) then $\Omega'(A, B)$ simply equals to $\Omega(A, B)$. If a pairing is possible then we first apply our algorithm recursively to the subgraphs rooted at N and M, i.e. we calculate the value $L = \Omega'(N, M)$. Next, we create DAGs A' and B' from the original ones by replacing N and M by single nodes, having no outgoing edges. The modified discrepancy is then calculated recursively as

$$\Omega'(A, B) = \Omega'(A', B') + L \tag{4}$$

The algorithm introduced above is summarised in Figure 6. In our implementation we offer the user a choice of the discrepancy function (Ω or Ω') through the graphical user interface (see Section 5.5).

```
Ω'(A,B) =
  if pairing_popular_nodes_possible(A,B) then
     (N,M) = pair_popular_nodes(A,B);
     A' = reduce(A, N);
     B' = reduce(B, M);
     return Ω'(A',B') + Ω'(N,M);
  else
     return Ω(A,B);
```

Figure 6: The DAG discrepancy algorithm with popular nodes.

5 The prototype implementation

In the following we present our test implementation of the framework, the *Match* plagiarism detection tool. The current version of Match supports two Front-ends,

[4]In our implementation (see Section 5) nodes with at least 10 incoming edges are considered popular.

for Prolog and SML, as we teach these two languages as part of a Declarative Programming course, and major assignments must be written in these languages. In this paper we describe the Prolog Front-end only, more about the SML interface can be read in [10].

In the following we discuss the implementation details of the relevant parts of the comparison framework (Front-end, Simplifier and Comparator), then we describe the graphical user interface of the system.

5.1 Mapping source code to a model

We chose *call graphs* as the models of Prolog programs. A call graph is a graph where the nodes correspond to the Prolog predicates and the edges to the calls. If in the body of predicate `A` there is a call to predicate `B` then an edge between the nodes `A` and `B` is present in the graph. When there are multiple calls, multiple edges are present. We decided to exclude the built-in predicates (such as `is/2`) from the graph, because they do very elementary tasks and would increase the graph size considerably, without increase in the precision of the model. The graph includes, however, the library predicates and also reflects implicit meta calls, made by using `findall/3`, for example.

Furthermore we made some simplifications to our model: we remove from the call graph the *self-loops*, which correspond to recursive calls. This is because explicit recursion is so common is Prolog that for us it does not contain valuable information. We also remove *back edges* (i.e. edges which point to an already visited node during a depth-first search) in order to avoid cyclic graphs, so that we can work with DAGs instead of general graphs. Although this means that simple reordering tricks can change the resulting graph (as they change the order in which a depth-first search visits the nodes) we do not consider this a problem. This is because in our model, circles actually correspond to so called mutual recursion (for example when `A` calls `B` and vice versa). Our experience, however, is that mutual recursion is very rarely used by students and so neglecting it does not effect the final similarity measure in a significant way.

Finally, those predicates to which there was no reference in the source code are not included in the graph, i.e. the call graph consists of a single component.

Call graphs are well suited for Prolog programs. This is because the only control structure of Prolog is the predicate invocation, it lacks `while`, `for`, `goto` or any other "usual" imperative control elements.

The call graph is built from the program source code by using static source code analysis. For this we slightly modified the `xref` package of SICStus Prolog.

5.2 Model reduction techniques

For the comparison of Prolog programs we have defined four reduction steps, which are applied in succession. This means that the comparison can be performed on five abstraction levels. To each reduction step we have assigned a similarity

factor as introduced in Figure 1. These factors were chosen empirically. The actual reduction steps and their similarity factors are the following:

1. *non-called predicates:* we remove those predicates which were not called during a test call. The test call is provided by the tester. Test calls are usually simple tests which are easily solvable by the programs. The assigned similarity factor is 0.95.
2. *library or dynamic predicates:* here we omit the library and dynamic predicates from the call graphs. The similarity factor is 0.9.
3. *multiple edges*: we remove multiple edges from the call graph and only keep one edge between any given two predicates. This helps handling the popular node problem presented in Section 4.3.3. We set the similarity factor to 0.8.
4. *topological isomorphism:* we remove those vertices from the call graphs which have a degree of 2 (one incoming and one outgoing edge). This helps to detect the call-tunneling trick. The similarity factor here is 0.7.

Actually the user of the plagiarism detection system can decide to skip some of the reductions steps (cf. Section 5.5), which results in less than 5 abstraction levels. In this case the factors can be different from the ones shown above, as the value of a factor is usually set in a "context sensitive" way, considering what other reduction steps have been done previously. For example, in our concrete implementation, if we use the first, second and fourth reduction step, but we do not filter multiple edges, the factor for the highest abstraction level (the one corresponding to step four above) is set to 0.85. Thus step 4 when applied after steps 1 and 2 is considered to be slightly less "destructive" than step 3 in the same context (the corresponding factors being 0.85 and 0.8, respectively). Our experience is that using step 3 and 4 simultaneously has significant cumulative effect, justifying the similarity factor of 0.7, when all the above reduction steps are applied.

5.3 Model comparison algorithms

In our system the comparison of models is based on the coding technique and the similarity degree introduced in Section 4.3. We have chosen this approach because we found that comparing codes often gives a good intuitive characterisation of the similarity of the programs.

For example, if the codes match the corresponding models are trivially the same. If one of the codes contains the other as its subsequence, then it can be suspected that one student got the other's program and added some new structure to it.

Call grouping can also be detected from the codes[5]. For example if predicate `P` calls `T` which calls four other predicates, the corresponding code will be `( L ( L L L L ) )`, where parentheses represent binary values and `L` represents a leaf as described in Section 4.3. If we apply call grouping, for example `T` will call `Q` and `W`, each of which will call two other predicates, then the code takes the form `( L ( (`

[5]We will use parenthesis in the codes below, instead of binary digits, as this makes call nesting more apparent.

`L L ) ( L L ) ) )`. Here the second, third, fourth and fifth parenthesis has to be removed in order to get the original code.

We use the widely available UNIX `diff` program to actually enumerate the differences between the codes A and B, i.e. to calculate $\Delta(A, B)$. The `diff` program uses a variation of the LCS algorithm (cf. Section 4.3.2). The way we make use of `diff` is the following. First we make two files corresponding to the two codes we would like to compare. The way we create the content of such a file is the following: each `(`, `)` and `L` in the tree code is put on a separate line. For example the file corresponding to the code `(LL)` shown in Figure 4 will contain four lines:

```
(
L
L
)
```

Next, we let the UNIX `diff` utility calculate how these files can be made equal, i.e. to produce the instructions on which leaves and nodes should be added or removed to make call graphs A and B isomorphic. Actually we always try to modify the bigger graph (i.e. the graph with longer code) and check what transformations we can use to obtain the smaller one.

By analysing the information given by the `diff` utility we assign a similarity degree to the pair of codes. As we described in Section 4.3, we start with degree 1 and for each difference we subtract a "penalty" fraction, which reflects how much we should "punish" the given modifications of the code sequences. This corresponds to calculating equation (3) in Section 4.3.2. We found that the penalties shown in Table 1 are very usable[6]:

Type of modification	Penalty
removal of a leaf	0.01
addition of a leaf	0.03
removal of a node	0.02
addition of a node	0.06

Table 1: Penalties used by the Match system.

Accordingly, if we need to remove one leaf from our bigger call graph to make it identical to the smaller one, then the similarity degree is 0.99. As one can see, addition is always penalised more than removal. This is because of our experience that cheating students usually try to copy and modify the work of a fellow student

[6]Note that the addition and removal of a node actually corresponds to two differences, one for the opening and the other for the closing parenthesis.

while keeping the original parts intact. This way the original program will be part of the resulting (bigger) program. So, our assumption is that in case of plagiarism the bigger graph can be reduced to the smaller one by applying node and edge removals.

According to this assumption, we actually offer to use the `diff` program in two different modes. The mode named `sdiff` (simple `diff`) means that we only consider those transformations that require only node or edge removals from the bigger graph. Otherwise we conclude that no plagiarism happened. In the other mode, called `fdiff` (full `diff`), we make no such assumption. This results in more false positive results, but it also increases recall significantly (see Section 6).

Having explained the concrete implementation, we reiterate the issue of abstraction levels. Let us consider two graphs which differ only in the multiplicity of the edges. In the absence of abstraction levels, using `diff` alone, we could easily get a similarity degree of 0, provided there is a sufficient number of multiple edges in the graph. However, when the multiple edge removal abstraction is applied we get a similarity of 0.9, which may be more appropriate. This shows that the introduction of abstraction levels is a useful extension, in addition to the `diff` algorithm.

We have also made a further improvement in the calculation of the similarity degrees, as discussed in the following subsection.

5.4 Generating mappings between predicates

Although by calculating the similarity degree of the source programs and presenting the user the most promising pairs we have already fulfilled our original goal, it greatly helps the user of the system if we present some kind of a "proof" of cheating as well. We produce such a "proof" in the form of a mapping between the predicates of the two programs. In this mapping a predicate of one program is paired with the predicate of the other program which is most similar to it. This is a very useful guide to the user when she verifies the results manually.

These mappings are generated by a systematic deterministic traversal of the codes in question. We start from the nodes corresponding to the root predicate (i.e. a predicate which is the entry point of all the student programs). These nodes are paired with each other. Then we visit the neighbours of the starting nodes and pair them using their codes. We continue this algorithms recursively. Whenever there is ambiguity, e.g. we are examining two nodes with multiple neighbours having the same codes, we use a heuristic: those nodes will be paired whose number of arguments differ the least. Note that the mappings are actually derived from the models belonging to the abstraction level where the maximal similarity degree was found. An example mapping is shown in Figure 8.

As mentioned above, the "quality" of the mapping is also taken into consideration when calculating the similarity degree. In the current implementation we actually decrease the similarity degree of the programs by 0.005 for each pair of predicates mapped to each other, which have different arities.

5.5 The graphical user interface

Match offers a graphical user interface (GUI) where the user can customise the parameters of the comparison and can view the results.

A screenshot of the main window can be seen in Figure 7 (it shows the state of the system right after a successful execution)[7]. On the top of the window we find four buttons. The first one called "Make info" invokes the Front-end, i.e. it creates the models from the source codes. This practically means that Match searches for source programs in the given directory and for each source code it creates a special file containing the labelled call graph.

These files are loaded by the second button named "Load info". In the bottom of the window we can see that in this specific example we loaded 32 such graphs.

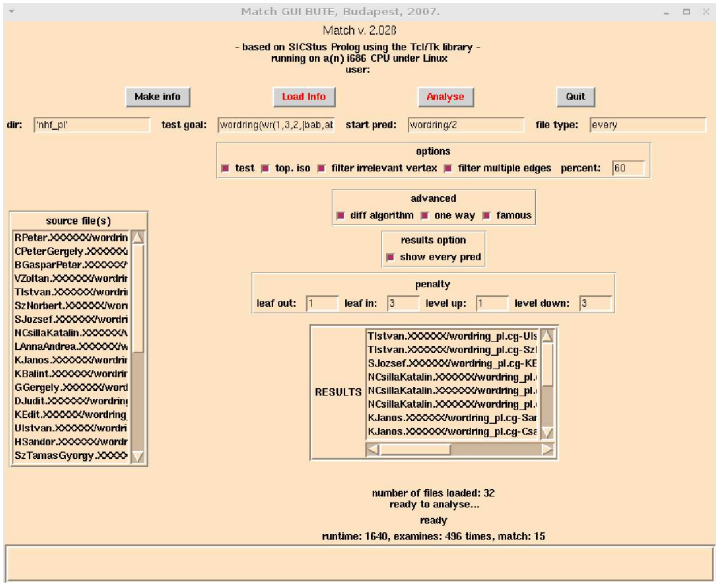


Figure 7: The Graphical User Interface of the Match system

The third button called "Analyse" starts the comparison process based on the parameters the user specified by using the check boxes located in the "options" and "advanced" areas. These options basically tell Match what kind of reduction steps it should apply, whether it should use the `diff` algorithm[8] and what is the similarity threshold (i.e. only hits with similarity degree greater than the threshold will be presented). In the example shown in Figure 7 we selected all the reduction steps,

[7] Note that we changed the name of the students because of privacy issues.

[8] If this option is disabled, then isomorphism is used instead of similarity, i.e. the similarity degree of the models is considered to be a binary value: 1 means that the models are isomorphic, 0 means they are not.

asked Match to use `diff`, take special care of the popular nodes (called famous in the GUI) and set the threshold to 60%.

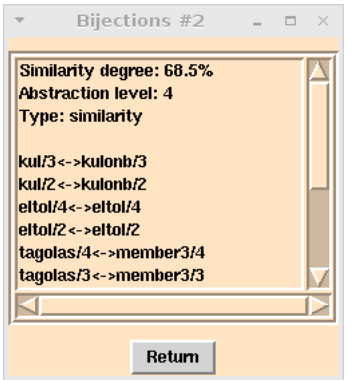


Figure 8: A mapping between predicates proving the fact of plagiarism

When we use the `diff` algorithm, we can also set the penalties (in units of 0.01) the Match program should use to determine the similarity of the two graphs. In this example the penalties are set to the ones described in Section 5.3.

After a successful execution the suspicious pairs of programs are shown in the middle of the screen under the title "Results". In Figure 7 we have 15 such pairs. If we select one of these pairs, Match displays the predicate mapping between the two source programs. Namely, we can see which predicate in one program matches which predicate in the other, as shown in Figure 8. In this concrete mapping we can see that, for example, predicate `kul` corresponds to predicate `kulonb`, both of them having 3 arguments. We can also see that the similarity of these programs is calculated to be 68.5%, and that this was found on abstraction level 4. The next line, "Type:", indicates that even on this abstraction level the codes were only "similar", i.e. non-isomorphic.

6 Evaluation

Below we first reiterate our goals presented in Section 3 and examine how they are fulfilled by the Match system. Next, we present real life execution results showing that the framework convincingly detects plagiarism in student programs.

Table 2 summarises the student tricks we have described in Section 3 and for each gives a brief explanation of how the given trick was handled in the implementation of the plagiarism detection framework.

We now proceed to discuss the performance evaluation of the Match system. We were lucky enough to have abundant amount of Prolog source programs to test the prototype on. Moreover, several students were kind enough to provide us with some hints on what cases were they cheating (*after* they completed the course and

Student trick	Preemptive measure
changing the names of the identifiers	we do not store names in our models
changing the arity of the functions	arity is not used in model comparison
splitting the program into several modules	module boundaries are not taken into account
reordering the function definitions	the call graph is not affected
reordering the statements within a function	the call graph is not affected
putting useless functions in the program	using the "non-called predicates" reduction step
putting useless calls in a function	using the "non-called predicates" reduction step
call-tunneling	using the "topological isomorphism" reduction step
call-grouping	using the `diff` algorithm

Table 2: Our goals and the way how they are achieved.

were promised full amnesty). So we had the minimal expectation that the Match system should at least mark those assignments as matched pairs.

The 73 source programs[9] were evaluated against 4 different similarity thresholds. For example, the 60% threshold means that our system shows pairs of source codes which have similarity degree at least 60% percent. For every threshold, the system was run with 18 different parameter variations. These include the most useful settings in practical cases. These 18 variations are based on the following 6 basic variants:

variant B:	all options are disabled (base case)
variant N:	filtering non called predicates
variant NL:	N + filtering library/dynamics predicates
variant NLM:	NL + filtering multiple edges
variant NLT:	NL + topological isomorphisms
variant NLMT:	NLM + topological isomorphisms

In the first six cases we do not use `diff`, i.e. we only consider graph isomorphism

[9]We had actually 92 submitted Prolog homework, but from these we excluded those programs that either do not compile or do not solve the required number of test cases.

between the models at different abstraction levels. The next twelve variations are obtained by applying `diff` in two different modes, simple and full (cf. Section 5.3).

	Execution time	Compared pairs	Hits	Relevant hits	Recall	Precision
B	15.40s	2628	6	6	46%	100%
N	20.33s	2628	6	6	46%	100%
NL	30.11s	2628	6	6	46%	100%
NLM	31.50s	2628	6	6	46%	100%
NLT	31.28s	2628	6	6	46%	100%
NLMT	33.40s	2628	16	8	61%	50%
B + sdiff	60.13s	2628	8	6	46%	75%
N + sdiff	78.07s	2628	8	6	46%	75%
NL + sdiff	124.41s	2628	17	8	61%	47%
NLM + sdiff	179.78s	2628	22	8	61%	36%
NLT + sdiff	162.34s	2628	17	8	61%	47%
NLMT + sdiff	364.04s	2628	32	10	76%	31%
B + fdiff	98.2s	2628	48	10	76%	21%
N + fdiff	122.02s	2628	48	10	76%	21%
NL + fdiff	180.78s	2628	65	11	84%	17%
NLM + fdiff	295.6s	2628	70	11	84%	16%
NLT + fdiff	232.60s	2628	65	11	84%	17%
NLMT + fdiff	414.43s	2628	80	13	100%	16%

Table 3: Match results for the threshold of 60%.

For every setting we measured the run time, the number of hits and the ratio of hits and the relevant hits (precision). To determine the relevant hits we examined the program pairs manually, and decided if the given case should be considered plagiarism or not. We also made a serious manual effort to check if there were any cases of plagiarism that were not found by Match. We concluded that the 13 pairs discovered by the most complex run of Match were the only cases where one program code was derived from the other one.

Assuming that the number of all hits is 13, we calculated the so called *recall* which is the ratio of the number of relevant hits returned by the program and the number of all hits.[10]

The results for threshold 60% are listed in Table 3. The test were run under Linux on Intel Celeron 450Mhz processor with 128 MByte of RAM.

From our tests we can draw several conclusions. First, we found that students do not often use sophisticated tricks. This can be seen from the fact that adding more reduction steps does not significantly improve the effectiveness of Match: most of the cheaters are caught already at the lower levels. At the same time further reduction steps do result in new hits, so higher abstraction levels are by no means useless. For example, 80% percent of the cheaters found in the second block were already uncovered after two reduction steps (variant NL). These included the programs of a pair of students who claimed they worked on the modifications for more than 5 hours, and in spite of this, their similarity degree was nearly 90%.

We can also see that although precision drops back significantly as we use more and more abstraction levels, the results are still acceptable. In the worst case (here the precision is 16%) one in six pairs of codes is a proper hit among 70 suspicious pairs. Although this requires some effort from the person verifying the results of Match, the amount of manual work is still almost two magnitudes less than that required when the plagiarism detection framework is not used (over 2500 cases).

We can also conclude that both the abstraction mechanism and the similarity degree calculated in (3) are needed. There were cases when the plagiarism was detected with a high degree of similarity due to the fact that only minor differences were found on abstraction level 1, for example. Without considering graph similarity, we would have needed to use more reduction steps to make these models isomorphic, resulting in a smaller similarity degree. This shows how useful the technique introduced in (3) can be. However, the opposite situation also occurred. Namely, we could find programs (relevant hits) which were isomorphic on a relatively high abstraction level, but calculating (3) on the previous level gave much lower similarity. This shows the significance of using reduction steps.

Our approach is very fast. For 73 programs, checking of all the pairs took between 15.4s (no diff and no abstraction levels) and 414s (when all of the reduction steps were applied and the full diff algorithm was used).

7 Future work

Our future plans include the integration of the most promising statistical and/or lexicographic approaches in the framework. This way we can use hybrid comparison techniques which we hope can be more efficient than the pure structural approach in some cases.

We would also like to develop Front-ends for further programming languages. In case of procedural languages, such as C, Pascal or several script languages, the

[10] Precision and recall are measures widely used for evaluation information retrieval techniques, see e.g. in [11].

control structures are the main bearers of information. When dealing with object-oriented languages, C++ or Java for example, the class hierarchies should also be properly represented in the models.

The heuristics used by the Match system can also be improved. These include the heuristics used when pairing popular nodes and predicates between two models. Furthermore, the user interface lacks some functionality. For example, it is often the case that a larger group of students submit similar assignments. In such a case Match returns all pairs within the group as suspicious. It would help the person verifying the results, if the group of cheaters was identified, as a whole.

Beside homeworks, we would also like to measure the performance of the framework for really large source programs (e.g. millions of lines of codes) as well.

Finally, it would be interesting to extend the Match system with adaptive penalty weights, i.e. to let the system automatically determine the penalty function based on certain properties of the given source programs (size, complexity, etc).

8 Conclusions

In the paper we have presented a plagiarism detection framework which is capable of calculating a similarity degree for a pair of program sources. The framework uses directed, labelled graphs to represent the structural information extracted from the programs. Instead of using sophisticated comparison algorithms our approach combines the use of relatively simple comparison techniques together with simplifying graph transformations, called reduction steps.

We have presented the three main components of the generic framework: the Front-end which converts programs to graphs, the Simplifier, which carries out the reduction steps and the Comparator, which calculates a similarity degree for the graphs. We have described the implementation of the framework, the Match system, which has been successfully used to detect plagiarism in homework assignments for years. We have also presented a detailed performance evaluation of the system.

We believe that the novel architecture of our approach, based on simplifying graph transformations and straightforward comparison algorithms, has proved to be a viable technology for plagiarism detection in source programs.

Acknowledgements

The authors would like to thank the help of Tamás Benkő in the development of the Match system.

References

[1] V. Arvind and Piyush P. Kurur. Graph isomorphism is in SPP. *Inf. Comput.*, 204(5):835–852, 2006.

[2] Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *In Proceedings of the 25th Annual Symposium on Theory of Computing*, pages 71–80, 1993.

[3] Brenda S. Baker and Udi Manber. Deducing similarities in Java sources from bytecodes. In *Proc. of Usenix Annual Technical Conf.*, pages 179–190, 1998.

[4] J.M. Bieman and N.C. Debnath. An analysis of software structure using generalized program graph. In *In Proceedings of COMPSAC*, pages 254–259, 1985.

[5] Computer and Automation Institute of the Hungarian Academy of Sciences. Online plagium detection portal. `http://www.kopi.sztaki.hu/`, 2007.

[6] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. In L. Brankovic and J. Ryan, editors, *Proceedings of the11th Australasian Workshop On Combinatorial Algorithms*, pages 75–86, Hunter Valley, Australia, 2000.

[7] David Eppstein. Subgraph isomorphism in planar graphs and related problems. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1995.

[8] J. A. W. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Comput. Educ.*, 11(1):11–19, 1987.

[9] Dick Grune. The software and text similarity tester sim. `http://www.cs.vu.nl/~dick/sim.html`.

[10] Dávid Hanák. Computer based support for teaching declarative languages (in hungarian). Master Thesis, 2001.

[11] David Hawking and Nick Craswell. Very large scale retrieval and web search. In Ellen Voorhees and Donna Harman, editors, *TREC: Experiment and Evaluation in Information Retrieval*. MIT Press, 2005.

[12] Paul Heckel. A technique for isolating differences between files. *Commun. ACM*, 21(4):264–268, 1978.

[13] Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.

[14] Gwyneth Hughes, Sharon Brown, Mike Jakobson, Chris Philpot, Paul Dwight-Moore, Nick Jarrett, Toby Grainger, and Barry Short. Report on the viability of copycatch plagiarism detection software. `http://www.copycatchgold.com/`, 2002.

[15] Digital Integrity. Findsame. `http://www.findsame.com/`, 2007.

[16] iParadigms LLC. iThenticate. `http://www.ithenticate.com/`, 2007.

[17] A. Jovanović and D. Danilović. A new algorithm for solving the tree isomorphism problem. *Computing*, 32(3):187–198, 1984.

[18] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[19] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[20] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[21] Seo-Young Noh, Sangwoo Kim, and S.K. Gaida. An XML plagiarism detection model for procedural programming languages. In *In Proceedings of the 2nd International Conference on Computer Science and its Applications*, pages 320–326, 2004.

[22] Seo-Young Noh, Sangwoo Kim, and Cheonyoung Jung. A lightweight program similarity detection model using XML and Levenshtein distance. In *The 2006 World Congress in Computer Science Computer Engineering, and Applied Computing*, pages 3–9, 2006.

[23] Alan Parker and James Hamblen. Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, 32(2):94–99, 1989.

[24] Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 100–109, Washington, DC, USA, 2004. IEEE Computer Society.

[25] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting, 2003. `http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf`.

[26] Narayanan Shivakumar and Hector Garcia-Molina. The SCAM approach to copy detection in digital libraries. *D-Lib Magazine*, 15, 1995.

[27] Jill Suarez and Allison Martin. Internet plagiarism: A teacher's combat guide. *Contemporary Issues in Technology and Teacher Education*, 1(4), 2001. `http://www.citejournal.org/vol1/iss4/currentpractice/article2.htm`.

[28] Adrian West. Coping with plagiarism in computer science teaching laboratories. Computers in Teaching Conference, Dublin, 1995.

[29] G. Whale. Identification of program similarity in large populations. *Comput. J.*, 33(2):140–146, 1990.

[30] Michael J. Wise. YAP3: Improved detection of similarities in computer program and other texts. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 28, 1996.

Received 18th July 2007