Detecting similarity of R functions via a fusion of multiple heuristic methods

Maciej Bartoszuk[1] **Marek Gagolewski**[2,3]

[1] Interdisciplinary PhD Studies Program, Systems Research Institute,
Polish Academy of Sciences, m.bartoszuk@phd.ipipan.waw.pl
[2] Systems Research Institute, Polish Academy of Sciences,
ul. Newelska 6, 01-447 Warsaw, Poland, gagolews@ibspan.waw.pl
[3] Faculty of Mathematics and Information Science, Warsaw University of Technology,
ul. Koszykowa 75, 00-662 Warsaw, Poland

Abstract

In this paper we describe recent advances in our R code similarity detection algorithm. We propose a modification of the Program Dependence Graph (PDG) procedure used in the GPLAG system that better fits the nature of functional programming languages like R. The major strength of our approach lies in a proper aggregation of outputs of multiple plagiarism detection methods, as it is well known that no single technique gives perfect results. It turns that the incorporation of the PDG algorithm significantly improves the recall ratio, i.e. it is better in indicating true positive cases of plagiarism or code cloning patterns. The implemented system is available as web application at http://SimilaR.Rexamine.com/.

Keywords: R, plagiarism and code cloning detection, fuzzy proximity relations, aggregation, program dependence graph, t-norms

1. Introduction

Finding plagiarism in computer language source codes is not a trivial task, as definitions of code similarity are very fuzzy in their very nature. Firstly, even if two pieces of code are very similar, or even the same, it cannot be treated as a 100% proof of that two students (a most often case when plagiarisms are considered is in programming classes) cheated. There can be some reasons, why two solutions are very similar: one is when a problem is simple and nature of the problem imposes the same solution. A second one is when task is formulated in such a way that it imposes the same solution. For example, some form of pseudocode is provided in a task's formulation.

That is why the problem should not be formulated as a plagiarism detection, but rather as a similarity code detection. The similarity of code can be formulated by mathematical formula in some way, while judging a plagiarism involves some familiarity with personal relationships between a group of students, level of skills of every student and his/her willingness to cheat the tutor. All such factors are of course hard to quantify.

Moreover, code similarity detection systems can be also used for code cloning detection. Code cloning happens when one person uses the same piece of code in many parts of a project, while some form of code refactoring should be used.

In a most general approach, two pieces of code are similar, when they calculate the same thing. But there is a problem with this approach, because all the correct solutions of a homework are similar to each other according to the definition. So another, more specific definition, can be formulated: two pieces of code are similar, when they calculate the same thing in the same way. But definition of "the same way" is very fuzzy and depends on a person and particular code. The same way of calculating can consist of using the same loops, the same functions calls and the same set of variables. But a cheating student can use some artificial, auxiliary variables, change one type of loop to another and call some other functions, which do the same. That is why the method has to classify "not so similar" functions as similar.

Each algorithm known from the literature, e.g. GPLAG [1], JPLAG [2], or MOSS [3], induces its own operational definition of code similarity. First of all, the problem with such algorithms is that they are not fitted well to the nature of functional programming languages like R, which is very popular among data analysts and statisticians. The second issue is that no single method gives perfect results in all the possible cases. In our approach, similarly as in [4], we calibrate and aggregate (using a statistical learning model) the output of four different algorithms. The intuition behind this approach is that a set of "weak classifiers" may perform better than individual ones. The novel idea is to make the similarity comparison results non-symmetric: for every pair of functions (f_i, f_j) it may happen that $\mu(f_i, f_j) \neq \mu(f_j, f_i)$, thus we do not measure directly the degree of $f_i \cap f_j$, but $f_i \setminus f_j$ and $f_i \setminus f_j$ separately. Then, the results may be symmetrized by using a t-norm, which induces an additional "degree of freedom", which then may be calibrated.

The paper is structured as follows. Sec. 2 establishes the notation used in this paper, formalizes the problem, and lists some typical plagiarism attacks. Sec. 3 briefly recalls the 3 methods of fuzzy proximity measures used to compare two functions' source codes in [4] and describes the fourth method based on a Program Dependence Graph with novel improvements. In Sec. 4 we present an empirical study of the algorithm's discrimination performance. Finally, Sec. 5 concludes the paper.

2. Method overview and problem formulation

Assume that we have n functions' source codes $\mathcal{F} = \{f_1, \ldots, f_n\}$, where f_i is a character string, i.e. $f_i \in \bigcup_{k=1}^{\infty} \Sigma^k$, Σ is a set of e.g. ASCII-encoded characters. Each f_i is properly normalized by i.a. removing unnecessary comments and redundant whitespaces and applying the same indentation style. Normalization is easy in R, as we can call `f <- deparse(parse(text=f))`.

Before any method of plagiarism detection can be created, there is a need to recognize what types of "attacks" can be performed. Below we include a taxonomy of plagiarism attacks in R language.

Easy

- Add/remove comments
- Change names of variables
- Change "<-" into "=" or "->"

Moderate

- Change the order of lines of code
- Add/remove line(s) of code
- Expand/shrink of function calls, e.g.:

```
x[order(unlist(lapply(x,f)))]
```

and

```
y <- unlist(lapply(x,f))
o <- order(y)
x[o]
```

Hard

- Change loop into its equivalent form (*for* into *while*, but also into *lapply*), e.g.:

```
y <- numeric(n)
k <- 1
for(i in x) {
    y[k] <- sqrt(i)
    k <- k+1
}
```

and

```
y <- unlist(lapply(x,
    function(element){
        return(sqrt(element))})
```

or even

```
y <- sqrt(x)
```

There are many methods of code similarity detection known in the literature, such as string-based, token-based [2, 3], or Program Dependence Graph-based [1]. Every method focuses on different features of code. In this paper we define 4 similarity measures $\mu_1, \ldots, \mu_4$, which take as arguments text representations of two functions f_i and f_j. Every method returns a value from 0 to 1, formally $\mu_k(f_i, f_j) \in [0; 1]$ for $k = 1, \ldots, 4$, how much f_i is similar to f_j. Value 1 indicates that one function is a proper subset of a second function, while 0 means that these two functions have nothing in common.

Standard approaches treat code similarity as an equivalence relation, while we propose a subset-type relation, which is not symmetric. Let us consider an example, where f_1:

```
s <- 0
for(i in x){s <- s + i}
```

and f_2:

```
s <- 0
for(i in x){s <- s + i}
m <- 0
for(i in x){m <- m*i}
```

We are rather interested in methods which return $\mu(f_1, f_2) = 1$ and, say, $\mu(f_2, f_1) = 0.5$.

3. Four code similarity measures

The usage of the first three methods was proposed in [4]. Let us briefly recall these methods and after that the fourth, new method (with proper modifications for the R language) is described in detail.

3.1. Edit distance

The first method is based on simply comparing plain-text of functions' source code. In the first version of our system we used the Levenshtein distance. Informally, the Levenshtein distance between two strings is the minimum number of single-character edits (i.e. insertions, deletions, or substitutions) required to change one string into the other one.

In the current version we use the `adist()` function from the TRE library with an argument `partial=TRUE`. This function is not symmetric, e.g.

```
adist("abcxxx", "abc", partial=TRUE)
```

returns 3, while

```
adist("abc", "abcxxx", partial=TRUE)
```

returns 0.

Our first method is defined as:

$$\mu_1(f_i, f_j) = 1 - \frac{\mathrm{dist}_{f_i,f_j}(|f_i|, |f_j|)}{|f_i|}$$

It is easily seen that for a pair of identical strings we obtain the value of 1. On the other hand, for "abc" and "defghi" we get 0, as $\mathrm{dist}_{\text{"abc"},\text{"defghi"}}(3, 6) = 3$.

3.2. Tokens

Methods based on tokens are quite popular. Two known tools, JPLAG [2] and MOSS [3] are based on tokens. The difference between our approach and the classic one is that we do not use a symmetric metrics. At the last step we divide common part of functions (generally it is a number of common tokens) by a number of tokens of one function, not the sum of tokens from two functions. The similarity of two token strings f_i' and f_j' is computed via:

$$\mu_2(f_i', f_j') = \frac{\text{coverage}(\textit{tiles})}{|f_i'|}.$$

For more details please refer to [4].

3.3. Function calls counts

The third method is very effective and dedicated to the R language. Let $\mathcal{R}$ denote the set of names of all possible R functions and $c_i(g)$ be equal to the number of calls of $g \in \mathcal{R}$ within f_i. This method is defined with:

$$\mu_3(f_i, f_j) = \frac{\sum_{g\in\mathcal{R}} \min(c_i(g), c_j(g))}{\sum_{g\in\mathcal{R}} (c_i(g))}.$$

Again, it is not a symmetric method and we obtain this property by dividing numerator by sum of function calls for one function only.

3.4. A method based on a Program Dependence Graph

The first three methods are not dedicated to cases when a plagiarist swaps two lines of code, change loop type (e.g. `for` loop into `while` loop, or, what is more complex, into `apply` function), or does something which we call function calls' "expanding/shrinking". For example, the following function:

```
sortlist <- function(x, f) {
    x[order(unlist(lapply(x, f)))]
}
```

is "expanded" into:

```
sortlist <- function(x, f) {
    v1 <- lapply(x, f)
    v2 <- unlist(v1)
    o  <- order(v2)
    x[o]
}
```

That is why we decided to implement a method based on the Program Dependence Graph (PDG), firstly introduced in [5]. One of the known antiplagiarism system based on PDG is GPLAG [1] with some modifications discussed in [6, 7].

The Program Dependence Graph is a graph, in which single expressions are nodes and there are two types of edges: control dependency and data dependency edges, see Figs. 5 and 6. The former tell us about loop and if statements structure. The subgraph of PDG, where there are control dependency edges only is called Control Dependence Subgraph (CDS). The latter tells us if an expression is getting data from another expression (e.g. we use variable a, so we use data from assignment to variable a), or is a source of data for another instructions. The subgraph of PDG, where there are data dependency edges only is called Data Dependence Subgraph (DDS). What is more, we will use term *control flow edges*, which are not a part of PDG, but are needed to construct DDS. *Control flow edges* just tell us about order of expressions (nodes).

3.4.1. Program Dependence Graph creation

In this section the process of creation the Program Dependence Graph (PDG) is described. Firstly, the Control Dependence Subgraph (CDS) has to be generated, and after that the Data Dependence Subgraph (DDS) creation is based on the CDS. The algorithm is strongly based on [8]. The reader can find more details in that technical report.

The algorithm has been implemented in C++ using the Rcpp package. This package makes possible to create C++ code chunks for R language. Our C++ implementation bases on the Boost Graph Library.

Control Dependence Subgraph Every vertex in the PDG has two properties: USES and GEN. USES are names of variables which are used in a corresponding statement. GEN is a variable's name which is created in a corresponding statement. For example, for "`i<-a+b`" statement, USES are {"`a`","`b`"} and GEN is "`i`".

The algorithm starts by creating an artificial vertex "Entry" in PDG. It is a vertex, on which every top level expression will be control dependent. After that, we call `CreateCDS()` (see Fig. 1) which iterates over every expression in a function. This function determines what is the type of an expression and calls appropriate helper function for it, see Fig. 2 for an example. For details, like dealing with `break` or `next` statements, we refer reader to [8]. The next paragraphs describe the changes we made to the algorithm, so it is more adjusted to the R language.

R is a functional language. It is the a very common practice to call one function and provide it as an argument to another function (compare to "expand/shrink" of function calls in Sec. 2). Our implementation ensures that such alterations result in the same PDG.

First of all we fetch the most nested call. We create a vertex in a PDG for it. USES are variables used as arguments in this call. GEN is some generated unique name. After that we substitute this most nested call with the generated unique name. A function call which gets this most nested call as an argument will have this generated unique name in its USES. Informally, we expand every call in such a way that we assign every call to a variable and after that we use these variables as an arguments in consequent calls (see "expanded version" in Sec. 2). The pseudocode for processing a function call is in Fig. 3.

Another modification is that in the R language there is another type of a loop: the `apply()` function. It is a function which takes a vector, list, matrix or data frame as an argument and applies a given function on every its element (or row or column for matrices and data frames). There is a family of those functions: `lapply()`, `mapply()`, `sapply()` etc. What is more, we cannot rely on the name of a function, as it can be very easily changed, e.g. `apply2 <- apply`.

That is why we implemented the following approach: we check if some argument of a function call is an anonymous function. If this is the case, we assume that it is an apply-like function and expand it as a normal loop, like `for`. Arguments to this anonymous function are the iterating variables (and we get them into GEN) and the another arguments of the apply-like function are the variables names for USES.

Before we can get further we also have to create control flow edges. A control flow edge connects two vertices if and only if two corresponding expressions immediately follow each other. Control flow edges in PDG enable to recreate the order of expressions in a function. For more information please refer to [8].

Data Dependence Subgraph After we obtain CDS, we can produce DDS. We have to introduce another properties to vertices of PDG: IN and OUT. These are dictionaries, where a key is a variable name (character string) and a value is a vertex index, where the variable name is generated. The pseudocode which produces DDS is listed in Fig. 4. What is important, we have to get predecessors of a vertex in a control flow subgraph (graph where there are only control flow edges).

One of the novel ideas was to create transitive data edges. For example, consider two chunks of code:

```
d <- (b + c) * e
fun(d)
```

and

```
d <- (b + c) * e
f <- d
fun(f)
```

A call to function `fun()` is data dependent on `d`, but in the second chunk of code it is dependent on `f`. We may add transitive data edges, so that if vertex B is dependent on A, and C is dependent on B, we add data edge from A to C. Unfortunately, adding these edges was computationally demanding and the results were worse than for a graph without them. Probably there were too many data edges and it was easier to find false isomorphisms.

```
createCDS(node n, vertex parent)
{
  for(node n1 in n.children()) {
    switch(TYPEOF(n1)) {
      case FOR:
         createForNode(n1,parent);
         break;
      case WHILE:
         createWhileNode(n1,parent);
         break;
      case REPEAT:
         createRepeatNode(n1,parent);
         break;
      case IF:
         createIfNode(n1,parent);
         break;
      case BREAK:
         createBreakNode(n1,parent);
         break;
      case NEXT:
         createNextNode(n1,parent);
         break;
      case ASSIGNMENT:
      case CALL:
         createCallNode(n1,parent);
         break;
    }
  }
}
```

Figure 1: Pseudocode of CDS building in general

```
createForNode(node n, vertex parent)
{
  //statement below creates new vertex
  //in program dependence graph
  forVertex = CreateNewVertex();
  CreateControlEdge(parent, forVertex);

  //in statement below for
  //"for(i in x)"
  //we get variable "i"
  forVertex.GEN =
     getIterationVariableFromFor(n);
  //in statement below for
  //"for(i in x)"
  //we get variable "x"
  forVertex.USES =
     getUsedVariablesFromFor(n);

  node n1 = getBodyOfFor(n);
  createCDS(n1, forVertex);
  //statement below makes proper edges
  //for "break" and "next" statements
  makeStructuredTransfers(forVertex);
}
```

Figure 2: Pseudocode of CDS building for `for` statement

```
createCallNode(node n, vertex parent)
{
  //every function which gets an
  //anonymous function as an argument
  //is expanded to a loop construct
  //like for or while
  if(isApplyFunction(n))
     createApplyLoop(n);

  //statement below creates new vertex
  //in program dependence graph
  callVertex = CreateCallVertex();
  CreateControlEdge(parent, callVertex);

  //statement below can approach another
  //function call as as an argument
  //and call createCallNode() for it again
  callVertex.USES = getAllCallArguments(n,
     parent);

  if(n is assignment)
     callVertex.GEN =
       n.leftVariableOfAssignment;
  else
     callVertex.GEN =
       createUniqueName();
}
```

Figure 3: Pseudocode of CDS building for function call statement

```
createDDS(Graph CDS)
{
  bool changes = true;
  while(changes) {
    changes = false;
    //use breadth first search
    foreach(vertex v in CDS) {
      v.IN = ∪ P.OUT,
        P is a control flow
          predecessor of v
        v.OUT = v.OUT ∪ v.IN
    }
    for(variableName in v.USES) {
      for(vertex v1 in
         v.OUT[variableName]) {
           CreateDataEdge(v1, v);
      }
    }
    v.OUT[v.GEN].insert(v);
    if (there is change in v.OUT)
      changes = true;
  }
}
```

Figure 4: Pseudocode of DDS building

3.4.2. Finding maximum common subgraph isomorphism

After we obtain PDGs, we have to compare them in some way, so we can decide, which ones are similar. GPLAG [1] and also its successors [6, 7] solve *subgraph isomorphism problem* using the VF algorithm [9] to decide whether there is a similarity between two PDGs of corresponding functions. Assume that there are two PDGs: G and G'. We find a subgraph S' of G' which is isomorphic to G. Of course small change in G causes that it is not isomorphic to any $S' \subseteq G'$. What is more, we have to test it in two ways: whether G whether a subgraph isomorphic to *G'* and also if *G'* is subgraph isomorphic to *G*. Authors of GPLAG solve this by introducing the so-called γ-isomorphism, which means that, S has to be a subgraph isomorphic to G', where $S \subseteq G$ and also $|S| \geq \gamma|G|$, $\gamma \in (0, 1]$. The use of $\gamma = 0.9$ is recommended.

We decided to propose a different approach. Assume that we need to evaluate how much objects A and B are similar to each other. In all similarity problems, there is a need to find common part of A and B, and after that to calculate, how large is this common part with respect to A and B. The answer to this problem may be provided by solving the *maximum common subgraph isomorphism problem* (MCS). Assume that we have again two graphs: G and G'. Solving MCS is answering the question "what is the largest subgraph of G isomorphic to a subgraph of G'?". We decided to use the McGregor algorithm [10], which is implemented in the C++ Boost library.

Of course, the *maximum common subgraph isomorphism problem* is NP-complete, as well as the subgraph isomorphism problem [11]. We observed that the McGregor algorithm is finding a subgraph near to the exact common part quite quickly. Our heuristic is to compute $\beta \max(|V(G)|, |V(G')|)$ iterations, where $\beta \geq 1$. In our study we have used $\beta = 5$.

Assume that we create a PDG G_i for every R function f_i we want to check plagiarism. Denote the common part of graphs G_i and G_j as a H_{ij}. We define the fourth method as:

$$\mu_4(f_i, f_j) = \frac{|V(H_{ij})|}{|V(G_i)|}$$
$$\mu_4(f_j, f_i) = \frac{|V(H_{ij})|}{|V(G_j)|}$$

Please note that $|V(H_{ij})|$ is calculated only once.

4. Experimental results

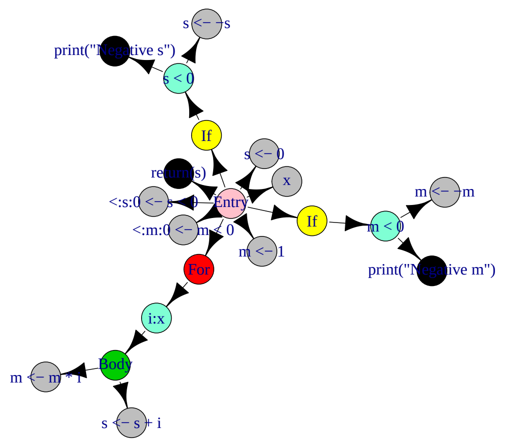
For a given similarity measure μ_k and a pair of functions (f_i, f_j) we get two degrees of similarity, $\mu_k(f_i, f_j)$ and $\mu_k(f_j, f_i)$. We aggregate these two values with a t-norm. A t-norm is a function $T\colon [0, 1] \times [0, 1] \to [0, 1]$ which satisfies the following properties:

- Commutativity: $T(a, b) = T(b, a)$
- Monotonicity: $T(a, b) \leq T(c, d)$ if $a \leq c$ and $b \leq d$

```
sum <- function(x)
{
  s <- 0
  m <- 1
  for(i in x) {
    s <- s + i
    m <- m*i
  }

  if(s < 0) {
    s <- -s
    print("Negative s")
  }
  if(m < 0) {
    m <- -m
    print("Negative m")
  }
  return(s)
}
```

(a) Source code of an R function

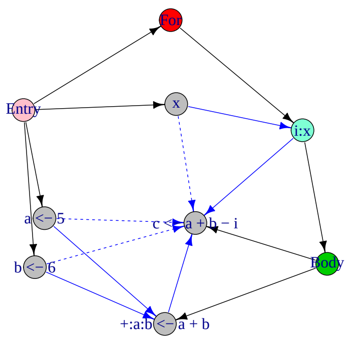


(b) Control Dependence Graph of an R function

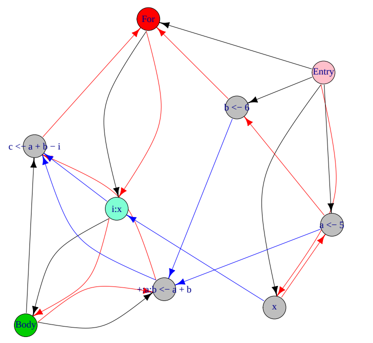
Figure 5: Example of transforming an R function into CDS

```
sum<-function(x)
{
  a <- 5
  b <- 6
  for(i in x)
  {
    c<-a+b-i
  }
}
```

(a) Source code of an R function



(b) Program Dependence Graph with dashed transitive data dependency edges



(c) Program Dependence Graph with control flow edges

Figure 6: Example of transforming R function into PDG

- Associativity: $T(a, T(b, c)) = T(T(a, b), c)$
- The number 1 acts as identity element: $T(a, 1) = a$

Among exemplary t-norms we find: minimum $T(a, b) = \min(a, b)$, product $T(a, b) = a \cdot b$, Łukasiewicz t-norm: $T(a, b) = \max(0, a + b - 1)$.

Now there is a need to properly aggregate and calibrate the methods' output. For that purpose, a random forest model is used. We created a learning set, where every observation (f_i, f_j) is represented as $T(\mu_k(f_i, f_j), \mu_k(f_j, f_i))$ for $k = 1, \ldots, 4$. Additionally, each such pair is classified so as to 0 denotes no similarity and 1 stands for a similar pair. As the process of acquiring learning data is time-consuming (see below), for the purpose of this experiment we decided to use an artificial set, which is a result of manual transformations of some predefined functions. This gives ca. 30 000 unique pairs of functions.

In order to obtain data for testing the performance of the proposed solution, we created a web application which is available at http://SimilaR.Rexamine.com/. Each user can create an account, send some group of files and asses the results. Of course, every expert has his/her own definition of code similarity and sometimes there are cases where one cannot be totally sure whether a pair of functions is suspicious or not. Thus, 5 grades of plagiarism can be chosen: *totally different*, *dissimilar*, *hard to say*, *similar* and *definitely similar*. At the time of writing this paper, ca. 400 pairs have been assessed.

In order to verify whether a fusion of multiple methods gives better results than individual methods, we decided to train a random forest model on the first learning set mentioned above and test it on the data from our website. We classified options *similar* and *definitely similar* as a plagiarism class and the rest as not plagiarism.

Table 1 summarizes the results. We show the performance of every individual method in the first 4 rows and after that we show the results for all the methods combined. In the study we chose the product t-norm, because it gave the best results for all the cases. The last 3 columns denote:

$$\begin{aligned} \text{accuracy} &= \frac{\text{TP}+\text{TN}}{\text{TP}+\text{TN}+\text{FP}+\text{FN}} \\ \text{recall} &= \frac{\text{TP}}{\text{TP}+\text{FN}} \\ \text{precision} &= \frac{\text{TP}}{\text{TP}+\text{FP}} \end{aligned}$$

where TP – number of true positives, i.e. actual similar pairs are found, FP – number of false positives, i.e. similarity is indicated, but it should not, TN – number of true negatives, i.e. no similarity indicated correctly, FN – number of false negatives, i.e. there is no similarity indicated, but it should.

We see that by aggregating the 4 methods we get a very high recall rate. This means that our system is able to correctly detect most of the similar pairs. In other words, if the output states that a pair is dissimilar, then it is highly possible that this is the case. On the other hand, still the precision rate should be improved – the system seems to be oversensitive and qualifies too many pairs as similar. This, however, may be due to the nature of our artificial training set.

Table 1: Comparison of performance of systems, product as a t-norm

Edit	**Tokens**	**F. calls**	**PDG**	**Acc.**	**Rec.**	**Prec.**
1	0	0	0	0.79	0.74	0.76
0	1	0	0	0.79	0.73	0.77
0	0	1	0	0.81	0.80	0.76
0	0	0	1	0.72	0.49	0.78
1	1	1	1	0.86	0.95	0.77

5. Conclusions

Our plagiarism detection system seems to be quite accurate. As far as the analyzed data set is concerned, it correctly classified most of the suspicious similarities. Moreover, the system is – at least theoretically – not vulnerable to typical attacks, like changing names of variables, substituting a `while` loop for a `for` loop, etc. We see that no single algorithm gives perfect results, but a proper data fusion leads to much better outcomes. Moreover, it turns out that making the individual similarity detection algorithms non-symmetric, and then symmetrizing them with a t-norm also improves the method's performance. This gives another "degree of freedom", which may be optimized and better fit data.

Our web application http://SimilaR.Rexamine.com/ serves as a tool not only for assessing the performance of our method, but also for gathering learning and test data. As soon as more data will be gathered, we will be able to calibrate our algorithms so that they give as good performance measures as possible.

For future work, we see a need for introducing some fingerprinting of R functions, so that only some pairs of them can be examined. This could increase the performance of the system. What is more, it would be possible to compare new R functions with whole set of R functions in our database. There is a *locality-sensitive hashing* (LSH) method, where hashing functions map similar keys to similar hash values. Halstead complexity measures also can be used for that purpose. These measures operates on the number of distinct operators, operands and total number of operators and operands.

Acknowledgments

This study was partially supported by the National Science Centre, Poland, research project 2014/13/D/HS4/01700.

Maciej Bartoszuk would like to acknowledge the support by the European Union from resources of the European Social Fund, Project PO KL "Information technologies: Research and their interdisciplinary applications", agreement UDA-POKL.04.01.01-00-051/10-00 via the Interdisciplinary PhD Studies Program.

References

[1] C. Liu, C. Chen, J. Han, and P.S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 872–881, New York, NY, USA, 2006. ACM.

[2] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[3] A. Aiken. Moss (measure of software similarity) plagiarism detection system. http://theory.stanford.edu/~aiken/moss/.

[4] M. Bartoszuk and M. Gagolewski. A fuzzy R code similarity detection algorithm. In Anne Laurent et al., editors, *Information Processing and Management of Uncertainty in Knowledge-Based Systems*, volume 444 of *Communications in Computer and Information Science*, pages 21–30. Springer, 2014.

[5] 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.

[6] W. Qu, Y. Jia, and M. Jiang. Pattern mining of cloned codes in software systems. *Information Sciences*, 259:544–554, 2014.

[7] W. Qu, M. Jiang, and Y. Jia. Software reuse detection using an integrated space-logic domain model. In *Proc. IEEE Intl. Conf. Information Reuse and Integration 2007*, pages 638–643, 2007.

[8] M.J. Harrold, B. Malloy, and G. Rothermel. Efficient construction of program dependence graphs. Technical report, ACM International Symposium on Software Testing and Analysis, 1993.

[9] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the VF graph matching algorithm. In *Proc. 10th Intl. Conf. Image Analysis and Processing*, ICIAP '99, page 1172, Washington, DC, USA, 1999. IEEE Computer Society.

[10] J.J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982.

[11] M.R. Garey and D.S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.