SimilaR: R Code Clone and Plagiarism Detection

by Maciej Bartoszuk and Marek Gagolewski

Abstract Third-party software for assuring source code quality is becoming increasingly popular. Tools that evaluate the coverage of unit tests, perform static code analysis, or inspect run-time memory use are crucial in the software development life cycle. More sophisticated methods allow for performing meta-analyses of large software repositories, e.g., to discover abstract topics they relate to or common design patterns applied by their developers. They may be useful in gaining a better understanding of the component interdependencies, avoiding cloned code as well as detecting plagiarism in programming classes.

A meaningful measure of similarity of computer programs often forms the basis of such tools. While there are a few noteworthy instruments for similarity assessment, none of them turns out particularly suitable for analysing R code chunks. Existing solutions rely on rather simple techniques and heuristics and fail to provide a user with the kind of sensitivity and specificity required for working with R scripts. In order to fill this gap, we propose a new algorithm based on a Program Dependence Graph, implemented in the **SimilaR** package. It can serve as a tool not only for improving R code quality but also for detecting plagiarism, even when it has been masked by applying some obfuscation techniques or imputing dead code. We demonstrate its accuracy and efficiency in a real-world case study.

Introduction

In recent years there has been a rise in the availability of tools related to code quality, including inspecting run-time memory usage (Serebryany et al., 2012), evaluating unit tests coverage (Ammann and Offutt, 2013), discovering abstract topics to which source code is related (Grant et al., 2012; Tian et al., 2009; McBurney et al., 2014; Linstead et al., 2007; Maskeri et al., 2008), finding parts of code related to a particular bug submission (Lukins et al., 2008), and checking for similarities between programs. With regards to the latter, quantitative measures of similarity between source code chunks play a key role in such practically important areas as software engineering, where encapsulating duplicated code fragments into functions or methods is considered a good development practice or in computing education, where any cases of plagiarism should be brought to a tutor's attention, see (Misic et al., 2016; Mohd Noor et al., 2017; Roy et al., 2009; Rattan et al., 2013; Ali et al., 2011; Hage et al., 2011; Martins et al., 2014). Existing approaches towards code clone detection can be classified based on the abstraction level at which they inspect programs' listings.

- *Textual* – the most straightforward representation, where a listing is taken as-is, i.e., as raw text. Typically, string distance metrics (like the Levenshtein one; see, e.g., van der Loo, 2014) are applied to measure similarity between pairs of the entities tested. Then some (possibly approximate) nearest neighbour search data structures seek matches within a larger code base. Hash functions can be used for the same purpose, where fingerprints of code fragments might make the comparison faster, see, e.g., (Johnson, 1993; Manber, 1994; Rieger, 2005). Another noteworthy approach involves the use of Latent Semantic Analysis (Marcus and Maletic, 2001) for finding natural clusters of code chunks.
- *Lexical* (token-based) – where a listing is transformed into tokens, which are generated by the parser during the lexical analysis stage. This form is believed to be more robust than the textual one, as it is invariant to particular coding styles (indentation, layout, comments, etc.). Typically, algorithms to detect and analyse common token sub-sequences are used (Kamiya et al., 2002; Ueda et al., 2002; Li et al., 2006; Wise, 1992; Prechelt et al., 2000; Hummel et al., 2011; Schleimer et al., 2003).
- *Syntactic* – an approach based on abstract syntax trees (ASTs). Contrary to the previous representation, which is "flat" by its nature, here the code is represented in a hierarchical manner. This makes it possible to, for instance, distinguish between a top-level statement and a code block that is executed *within* the `else` clause of an `if-else` construct. One way to quantify the similarity of ASTs is to find common sub-trees. This might be an uneasy task, therefore, e.g., in (Baxter et al., 1998) a hash function is used to project a sub-tree into a discrete single value, so that only the sub-trees in the same bucket are compared against each other. Another way involves the computation of diverse tree metrics (based on node type count, their distribution, connectivity, etc.) so that each AST is represented as a feature vector. Then the feature vectors

can be compared against each other directly in a pairwise fashion (Mayrand et al., 1996; Patenaude et al., 1999; Fu et al., 2017) or by means of some cluster analysis-based approach (Jiang et al., 2007).

- *Semantic* – the most sophisticated representation involving a set of knowledge-based, language-dependent transformations of a program's abstract syntax tree. Usually, a data structure commonly known as a Program Dependence Graph (PDG) is created, see below for more details. In such a data structure, the particular order of (control- or data-) *independent* code lines is negligible. A popular approach to measure similarity between a pair of PDGs concerns searching for (sub)isomorphisms of the graphs, see (Komondoor and Horwitz, 2001; Liu et al., 2006; Qu et al., 2014).

There are a few generally available software solutions whose purpose is to detect code clones, e.g., MOSS (see `http://theory.stanford.edu/~aiken/moss/` and Schleimer et al., 2003) and JPlag (see `http://jplag.de/` and Prechelt et al., 2000), see also (Misic et al., 2016; Vandana, 2018) for an overview. These tools are quite generic, offering built-in support for popular programming languages such as Java, C#, C++, C, or Python.

Unfortunately, there is no package of this kind that natively supports the R language, which is the GNU version of S (see, e.g., Becker et al., 1998; Venables and Ripley, 2000). It is a serious gap: R is amongst the most popular languages[1], and its use has a long, successful track record, particularly with respect to all broadly-conceived statistical computing, machine learning, and other data science activities (Wickham and Grolemund, 2017). With some pre-processing, MOSS and JPlag *can* be applied on R code chunks, but the accuracy of code clones detection is far from optimal. This is due to the fact that, while at a first glance being an imperative language, R allows plenty typical functional constructs (see the next section for more details and also, e.g., Chambers, 1998; Wickham, 2014; Chambers, 2008). On the one hand, its syntax resembles that of the C language, with curly braces to denote a nested code block and classical control-flow expressions such as `if..else` conditionals, or `while` and `for` (for each) loops. On the other hand, R's semantics is based on the functional Scheme language (Abelson et al., 1996), which is derived from Lisp. Every expression (even one involving the execution of a `for` loop) is in fact a call to a function or any combination thereof, and each function is a first-class object that (as a rule of thumb) has no side effects. Moreover, users might choose to prefer applying *Map–Filter–Reduce*-like expressions on container objects instead of the classical control-flow constructs or even mix the two approaches. Also, the possibility of performing the so-called *nonstandard evaluation* (metaprogramming) allows to change the meaning of certain expressions during run-time. For instance, the popular forward-pipe operator, `%>%`, implemented in the **magrittr** (Bache and Wickham, 2014) package, allows for converting a pipeline of function calls to a mutually nested series of calls.

In this paper we describe a new algorithm that aims to fill the aforementioned gap (based on Bartoszuk, 2018). The method's implementation is included in the **SimilaR**[2] package. It transforms the analysed code base into a Program Dependence Graph that takes into account the most common R language features as well as the most popular development patterns in data science. Due to this, the algorithm is able to detect cases of plagiarism quite accurately. Moreover, thanks to a novel, polynomial-time approximate graph comparison algorithm, its implementation has relatively low run-times. This enables to conduct an analysis of a software repository whose size is significant.

This paper is set out as follows. First we introduce the concept of a *Program Dependence Graph* along with its R language-specific customisations. Then we depict a novel algorithm for quantifying similarity of two graphs. Further on we provide some illustrative examples for the purpose of showing the effects of applying particular alterations to a Program Dependence Graph. What is more, we demonstrate the main features of the **SimilaR** package version 1.0.8. Then we perform an experiment involving the comparison of the complete code-base of two CRAN packages.

Program Dependence Graph

A *Program Dependence Graph* (PDG) is a directed graph representing various relations between individual expressions in a source code chunk. As we mentioned in the introduction, it is among the most sophisticated data structures used for the purpose of code clones detection. First proposed by Ferrante et al. (1987), it forms the basis of many algorithms, see, e.g., (Liu et al., 2006; Qu et al., 2014; Gabel et al., 2008; Krinke, 2001; Horwitz and Reps, 1991; Komondoor and Horwitz, 2001; Ghosh and Lee, 2018; Nasirloo and Azimzadeh, 2018).

[1]For instance, the 2018 edition of the *IEEE Spectrum* ranking places R on the No. 7 spot, see `http://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages`.

[2]See `https://CRAN.R-project.org/package=SimilaR`. **SimilaR** can be downloaded from the Comprehensive R Archive Network (CRAN) repository (Silge et al., 2018) and installed via a call to `install.packages("SimilaR")`.

Abstract Syntax Tree. To create a PDG, we first need to construct an *Abstract Syntax Tree* (AST) of a given program. In R, it is particularly easy to compute the AST corresponding to any expression, due to its built-in support for *reflection* that facilitates metaprogramming. For instance, the `parse()` function can be called to perform lexical analysis of a code fragment, yielding a sequence of language objects. Moreover, a basic version of a function to print an AST takes just few lines of code:

```
R> show_ast <- function(x) {
+      as.list_deep <- function(x) # convert to a plain list (recursively)
+          { if (is.call(x)) lapply(as.list(x), as.list_deep) else x }
+      x <- substitute(x)   # expression that generated the argument
+      str(as.list_deep(x)) # pretty-print
+  }
```

Let us visualise the AST corresponding to expression `d <-sum((x-y)*(x-y))`.

```
R> show_ast(d <- sum((x-y)*(x-y)))

List of 3
 $ : symbol <-
 $ : symbol d
 $ :List of 2
  ..$ : symbol sum
  ..$ :List of 3
  .. ..$ : symbol *
  .. ..$ :List of 2
  .. .. ..$ : symbol (
  .. .. ..$ :List of 3
  .. .. .. ..$ : symbol -
  .. .. .. ..$ : symbol x
  .. .. .. ..$ : symbol y
  .. ..$ :List of 2
  .. .. ..$ : symbol (
  .. .. ..$ :List of 3
  .. .. .. ..$ : symbol -
  .. .. .. ..$ : symbol x
  .. .. .. ..$ : symbol y
```

In R, both a constant (numeric, logical, string, etc.) and a symbol (name) constitute what we call a *simple expression*. A *compound expression* is in turn a sequence of $n+1$ expressions (simple or compound ones) $\langle f, a_1, \ldots, a_n \rangle$, $n \geq 0$, which represents a call to f with arguments $a_1, \ldots, a_n$ (which we are typically used to denote as $f(a_1, \ldots, a_n)$).

The above AST can be written in the Polish (prefix) notation as

$$\langle \text{`<-`}, \mathtt{d}, \langle \mathtt{sum}, \langle \text{`*`}, \langle \text{`-`}, \mathtt{x}, \mathtt{y} \rangle, \langle \text{`-`}, \mathtt{x}, \mathtt{y} \rangle \rangle \rangle \rangle$$

Such a notation is used in Scheme and Lisp; we skipped a call to '(' for readability, as `(e)` is equivalent to `e` for each expression `e`. Alternatively, the above can be written as

```
‘<-’(d, sum(‘*‘(‘-‘(x, y), ‘-‘(x, y))))
```

in the "functional" form. Let us emphasise that even an application of a binary operator (here: `<-`, `*`, and `-`) corresponds to some function call. Hence `x-y` is just a syntactic sugar for `` `-`(x,y)``. Moreover, other expressions such as `if..else` and loops also correspond to some function calls. For example:

```
R> show_ast(for(i in 1:5) {
+      print("i = ", i)
+      if (i %% 2 == 0) print(":)") else print(":$")
+  })

List of 4
 $ : symbol for
 $ : symbol i
 $ :List of 3
  ..$ : symbol :
  ..$ : num 1
  ..$ : num 5
```

```
$ :List of 3
 ..$ : symbol {
 ..$ :List of 3
 .. ..$ : symbol print
 .. ..$ : chr "i = "
 .. ..$ : symbol i
 ..$ :List of 4
 .. ..$ : symbol if
 .. ..$ :List of 3
 .. .. ..$ : symbol ==
 .. .. ..$ :List of 3
 .. .. .. ..$ : symbol %%
 .. .. .. ..$ : symbol i
 .. .. .. ..$ : num 2
 .. .. ..$ : num 0
 .. ..$ :List of 2
 .. .. ..$ : symbol print
 .. .. ..$ : chr ":)"
 .. ..$ :List of 2
 .. .. ..$ : symbol print
 .. .. ..$ : chr ":$"
```

Vertex and edge types. The *vertices* of a PDG represent particular expressions, such as a variable assignment, a function call or a loop header. Each vertex is assigned its own *type*, reflecting the kind of expression it represents. The comprehensive list of vertex types for the R language code-base used in **SimilaR** is given in Table 1. The number of distinct types is a kind of compromise between the algorithm's sensitivity and specificity. It was set empirically based on numerous experiments (Bartoszuk, 2018).

We may also distinguish two types of edges: *control dependency* and *data dependency* ones. The former represents the branches in a program's control flow that result in a conditional execution of expressions such as `if-else`-constructs or loops. A subgraph of a PDG consisting of all the vertices and only the *control dependency* edges is called a *Control Dependence Subgraph* (CDS).

The latter edge vertex type is responsible for modelling data flow relations: there is an edge from a vertex v to a vertex u, whenever a variable assigned in the expression corresponding to v is used in the

Id	Color	Type	Description
0	Olive	`Entry`	marks the beginning of a function
1	Light yellow	`Header`	loop
2	–	`Next`	`next`
3	–	`Break`	`break`
4	Orange	`If`	a conditional expression
5	Light blue	`If_part`	an expression to execute conditionally
6	Gray	`assignment`	an assignment expression, e.g., `name <- val`
7	Violet	`parameter`	a function parameter
8	–	`oneBracketSingle`	an expression involving `[i]`, e.g., `vector[-1]`
9	–	`oneBracketDouble`	an expression involving `[i,j]`
10	–	`oneBracketTripleOrMore`	an expression involving `[i,j,k,...]`
11	–	`twoBrackets`	an expression involving `[[i]]`, e.g., `list[["nameditem"]]`
12	–	`dollar`	an expression involving `$`, e.g., `df$column`
13	–	`funNoArguments`	a function call with no actual parameters
14	Red	`funOneArgument`	a function call with one actual parameter
15	–	`funTwoArguments`	a function call with two actual parameters
16	–	`funThreeArguments`	a function call with three actual parameters
17	–	`funFourOrMoreArguments`	a function call with four or more actual parameters
18	–	`stopifnot`	a call to `stopifnot()`
19	–	`logicalOperator`	a logical operator-based expression, e.g., `&`, `|` or `!`
20	Green	`arithmeticOperator`	an arithmetic operator-based expression, e.g., `+`, `-` or `*`
21	Blue	`comparisonOperator`	a comparison operator-based expression, e.g., `==`, `<=` or `<`
22	Green	`return`	a call to `return()`
23	Cyan	`colon`	a colon operator-based expression, e.g., `from:to`
24	Dark green	`symbol`	a symbol (name)
25	Dark green	`constant`	a constant value, e.g., `1`, `"string"` or `NA`

Table 1: Assumed Program Dependence Graph vertex types for the R language.

computation of the expression related to u. A spanning subgraph of a PDG that consists solely of the *data dependency* edges is called a *Data Dependence Subgraph* (DDS).

Hence, a PDG of a function `F()` is a vertex- and edge-labelled directed graph $F = (V_F, E_F, \zeta_F, \xi_F)$, where V_F is the set of its vertices, $E_F \subseteq V_F \times V_F$ denotes the set of edges ($(v, u) \in E_F$ whenever there is an edge from v to u), $\zeta_F : V_F \to \{\texttt{Entry}, \texttt{Header}, \ldots, \texttt{constant}\} = \{0, \ldots, 25\}$ gives the type of each vertex and $\xi_F : E_F \to \{\texttt{DATA}, \texttt{CONTROL}\}$ marks if an edge is a data- or control-dependency one. Note that each PDG is rooted – there exists one and only one vertex v with indegree 0 and $\zeta_F(v) = \texttt{Entry}$.

Example code chunks with the corresponding dependence graphs are depicted in Figures 1 and 2. The meaning of vertex colors is explained in Table 1.

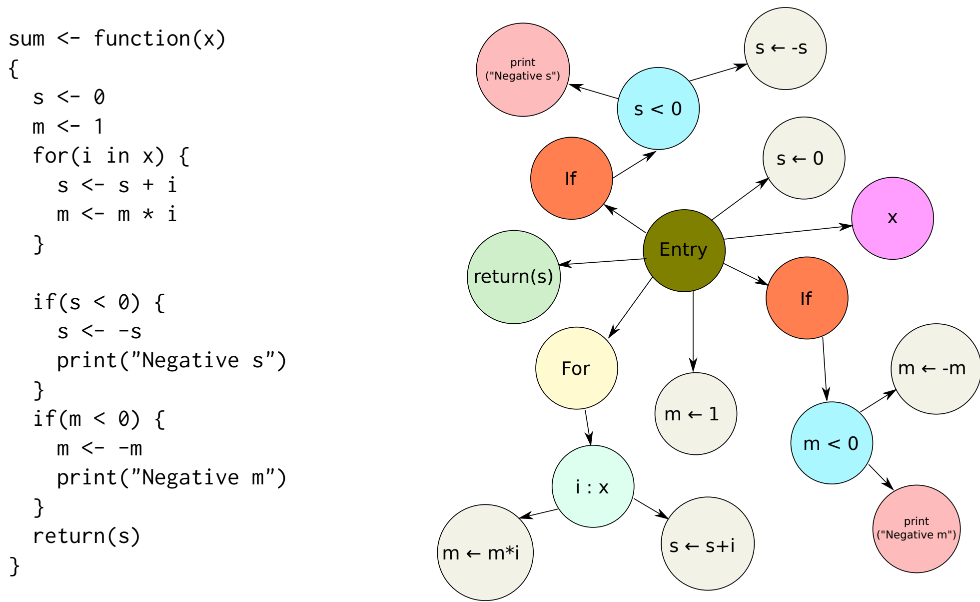

Figure 1: An example function and the respective Control Dependence Subgraph.

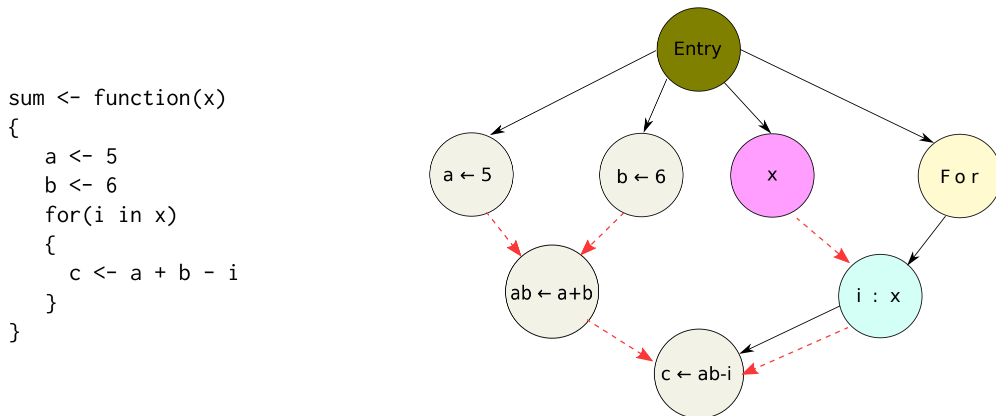

Figure 2: An example function and the corresponding Program Dependence Graph; solid and dashed arrows represent control and data dependency edges, respectively.

The most basic version of an algorithm to create a PDG based on an abstract syntax tree is described in (Harrold et al., 1993). Let us note the fact that a CDS is a subgraph of an AST: it provides the information about certain expressions being nested within other ones, e.g., that some assignment is part (child) of a loop's body. Additionally, an AST includes a list of local variables and links them with expressions that rely on them. This is a crucial piece of information used to generate DDS.

Note that, however, a PDG created for the purpose of code clones detection cannot be treated as a straightforward extension of a raw AST. The post-processing procedure should be carefully customised taking into account the design patterns and coding practices of a particular programming language. Hence, below we describe the most noteworthy program transforms employed in the **SimilaR** package so that it is invariant to typical *attacks*, i.e., transforms changing the way the code is written yet not affecting its meaning.

Unwinding nested function calls. As mentioned above, in R, as in any functional language, functions play a key role. A code chunk can be thought of as a sequence of expressions, each of which is composed of function calls. Base or external library functions are used as a program's building blocks and often very complex tasks can be written with only few lines of code.

For instance, given a matrix $\mathbf{X} \in \mathbb{R}^{d\times n}$ representing n vectors in $\mathbb{R}^d$ and a vector $\mathbf{y} \in \mathbb{R}^d$, the closest vector in $\mathbf{X}$ to $\mathbf{y}$ with respect to the Euclidean metric can be determined by evaluating `X[,which.min(apply((X-y)^2,2,sum))]`. This notation is very concise and we can come up with many equivalent forms of this expression written in a much more loquacious fashion.

Therefore, in **SimilaR**, hierarchies of nested calls, no matter their depth, are always recursively unwound by introducing as many auxiliary assignments as necessary. For instance, `f(g(x))` is decomposed as `gx <- g(x); f(gx)`. This guarantees that all their possible variants are represented in the same way in the PDG.

Forward-pipe operator, `%>%` Related to the above is the **magrittr**'s forward-pipe operator, `%>%`, which has recently gained much popularity within the R users' community. Even though the operator is just a syntactic sugar for forwarding an object into the next function call/expression, at the time of writing of this manuscript, the package has been used as a direct dependency by over 700 other CRAN packages. Many consider it very convenient, as it mimics the "left-to-right" approach known from object-orientated languages like Java, Python or C++. Instead of writing (from the inside and out) `f(g(x),y)`, with **magrittr** we can use the syntax `x %>% g %>% f(y)` (which would normally be represented as `x.g().f(y)` in other languages). To assure proper similarity evaluation, **SimilaR** unwinds such expressions in the same manner as nested function calls.

Calls within conditions in control-flow expressions. An expression serving as a Boolean condition in an `if` or a `while` construct might be given as a composition of many function calls. The same might be true for an expression generating the container to iterate over in a `for` loop. PDG vertices representing such calls are placed on the same level as their corresponding control-flow expressions so that they can be unwound just as any other function call.

Canonicalization of conditional statements. The following code chunk:

```
if (cond) {
  return(A)
} else {
  return(B)
}
```

is semantically equivalent to:

```
if (cond)
  return(A)
return(B)
```

This exploits the fact that the code after the `if` statement is only executed whenever the logical condition is false. To avoid generating very different control dependencies, we always unwind the former to the latter by putting the code-richer branch outside of a conditional statement.

Tail call to `return()`. The return value that is generated by evaluating a sequence of expressions wrapped inside curly braces (the `` `{`() `` function) is determined by the value of its last expression. If a function's body is comprised of such a code block, the call to `return()` is optional if it is used in the last expression. However, many users write it anyway. Therefore, a special vertex of type `return()` have been introduced to mark an expression that generates the output of a function.

***Map*-like functions.** Base R supports numerous *Map*-like operations that are available in many programming languages. The aim of the members of the `*apply()` family (`apply()`, `lapply()`, `sapply()`, etc.) is to perform a given operation on each element/slice of a given container. These are unwound as an expression involving a `for` loop. For instance, a call to `ret <-lapply(l,fun,...)` can be written as

```
ret <- list(); for (el in l) ret[[length(ret)+1]] <- fun(el, ...)
```

Variable duplication. To prevent redundant assignments such as `xcopy <-x` made just in order to refer to the original value under a new alias, a hierarchical variable dictionary is kept to generate *data dependency* edges properly.

Memoization. In pure functional languages it is assumed that functions have no side effects, i.e., the same arguments are mapped to the same return value. In R it is of course not always technically true (e.g., when the pseudo-random number generator is involved), but such an assumption turns out to be helpful in our context. Therefore, if a function call instance is invoked more than once, its value is memorised by introducing a new variable.

Dead code. Many plagiarism detection algorithms can be easily misled by adding random code that does not affect the main computations. In **SimilaR**, such *dead code* is identified and removed. This is done by iteratively deleting all vertices whose outdegree is zero (except those of type `return`).

To sum up, **SimilaR** guarantees that the Program Dependence Graph is the same regardless of the order of independent function calls, unwinding nested function calls, the use of the forward-pipe operator, etc. Hence, it is invariant to the most typical attacks. Moreover, it has been implemented in such a way that new kinds of transformations can be easily added in the future, as the R development practices and common program design patterns evolve.

Comparing Program Dependence Graphs

In our setting, code similarity assessment reduces to a comparison between a pair of Program Dependence Graphs. In this section we are interested in an algorithm μ such that $\mu(F, G) \in [0, 1]$ represents a similarity degree between two PDGs F and G. A similarity of 1 denotes that two PDGs are identical, while 0 means that they are totally different. Alternatively, we might be interested in a non-symmetric measure $\tilde{\mu}(F, G) \in [0, 1]$ representing the degree to which the source code of F is *contained* within G.

Ideally, an algorithm to compare two PDGs should enjoy the following properties:

- it should be *flexible* in the sense that introducing a "small difference" in one of the graphs should not affect the estimated similarity degree significantly;
- it should be *fast* to execute so that computing numerous pairwise similarities can be performed in a reasonable time span.

Due to the latter, we immediately lose our interest in all currently known *exact* algorithms to find subgraph isomorphisms or maximum common subgraphs because of their exponential-time complexity (the problems are NP-hard; see, e.g., Wegener, 2005). To recall, two graphs are isomorphic whenever there exists a mapping between the two graphs' vertices preserving the node adjacencies.

In the **SimilaR** package, we use a modified version (for increased flexibility and better performance in the plagiarism detection problem) of an algorithm described in (Shervashidze et al., 2011), which itself is based on the Weisfeiler–Lehman isomorphism test (Weisfeiler and Lehman, 1968) and graphs kernels. Note that the base method has been successfully used in many applications, e.g., in cheminformatics (Mapar, 2018) and programming autonomous robots (Luperto and Amigoni, 2019).

In each of the h iterations of the SimilaR algorithm, we assign new labels to the vertices of a PDG based on their neighbours' labels. While in the original algorithm (Shervashidze et al., 2011), two vertices are considered diverse already when one of their neighbours has been assigned a different

label, here we might still be assigning the same label if the vertices' adjacency differs only slightly. Our approach turns out to be more robust (Bartoszuk, 2018) against minor code changes or some vertices being missing in the graph.

SimilaR algorithm at a glance. Before we describe every step of the algorithm in detail, let us take a look at it from a bird's-eye perspective. If we are to assign each vertex a new label that is uniquely determined by their current type as well the labels allocated to their neighbours, two identical graphs will always be coloured the same way, no matter how many times we reiterate the labelling procedure. In particular, after h iterations, a vertex's label depends on the types of vertices whose distance from it is at most h.

We are of course interested in assigning equivalent labels to vertices in graphs that are not necessarily identical, but still *similar* to each other. Otherwise, two vertices which have all but one neighbour in common, would get distinct labels. After h iterations, all the vertices at distance at most h would all already be assigned different colours. This would negatively affect the overall graph similarity assessment.

In order to overcome this problem, we introduce the concept of vertex *importance*, which is based upon the number of vertices that depend on a given node. Only *important enough* differences in the vertex neighbourhoods will be considered as sufficient to trigger a different labelling. Then, after h iterations, two vectors of label type counts can be compared with each other to arrive at a final graph similarity degree.

SimilaR algorithm in detail. The following description of the SimilaR algorithm will be illustrated based on a comparison between two functions, `clamp1()` (whose PDG from now on we will denote with $F = (V_F, E_F, \zeta_F, \xi_F)$, see Fig. 3) and `standardise()` (denoted $G = (V_G, E_G, \zeta_G, \xi_G)$, see Fig. 4).

v	*Entry*	*x*	*min(x)*	*max(x)*	*- (1)*	*<*	*If*	*If_part*	*NULL*	*- (2)*	*/*
raw $\delta_F(v)$	7.80	3.04	1.35	1.12	0.93	0.65	0.30	0.20	0.10	0.21	0.10
normalised	0.49	0.19	0.09	0.07	0.06	0.04	0.02	0.01	0.01	0.01	0.01

Table 2: Vertex importance degrees in `clamp1()` (sum = 15.79, median = 0.04).

v	*Entry*	*x*	*sd(x)*	*<*	*If*	*If_part*	*NULL*	*mean(x)*	*-*	*/*
raw $\delta_G(v)$	4.33	1.71	0.93	0.65	0.30	0.20	0.10	0.33	0.21	0.1
normalised	0.49	0.19	0.10	0.07	0.03	0.02	0.01	0.04	0.02	0.01

Table 3: Vertex importance degrees in `standardise()` (sum = 8.86, median = 0.03).

v	*Entry*	*x*	*min(x)*	*max(x)*	*- (1)*	*<*	*If*	*If_part*	*NULL*	*- (2)*	*/*
$\zeta_F(v)$	0	24	14	14	20	21	4	5	25	20	20
$\zeta_F^1(v)$	0	1	2	2	3	4	5	6	7	3	3
$\zeta_F^2(v)$	7	1	2	2	8	3	4	5	0	6	6
$\zeta_F^3(v)$	8	5	6	6	7	1	2	3	0	4	4

Table 4: `clamp1()`: Labels assigned to vertices in each iteration.

v	*Entry*	*x*	*sd(x)*	*<*	*If*	*If_part*	*NULL*	*mean(x)*	*-*	*/*
$\zeta_G(v)$	0	24	14	21	4	5	25	14	20	20
$\zeta_G^1(v)$	0	1	2	4	5	6	7	2	3	3
$\zeta_G^2(v)$	7	1	9	3	4	5	0	2	6	6
$\zeta_G^3(v)$	12	9	10	1	2	3	0	11	4	4

Table 5: `standardise()`: Labels assigned to vertices in each iteration.

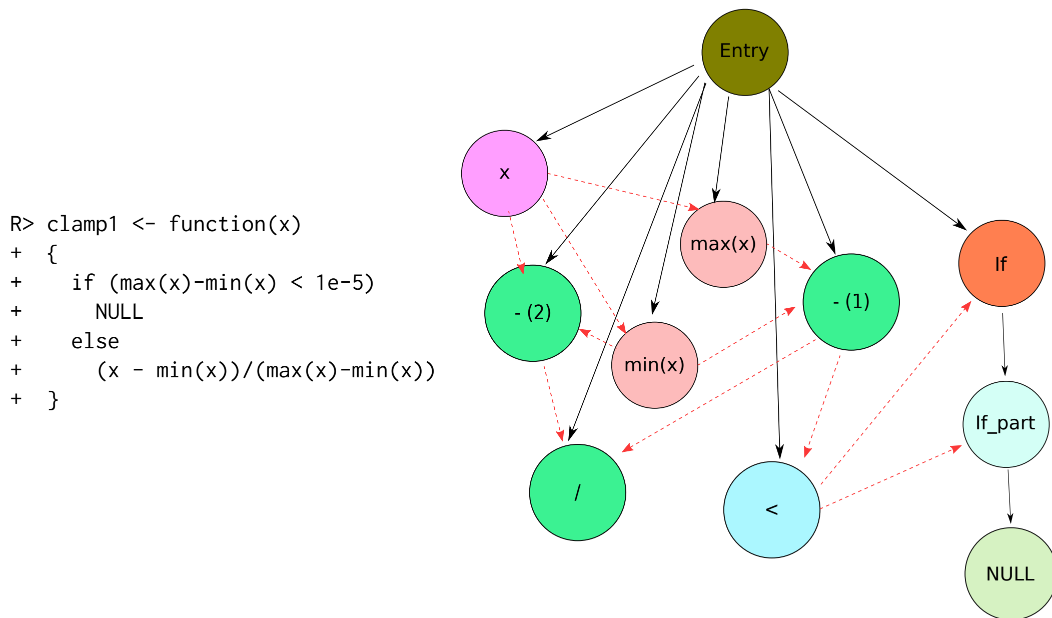

Figure 3: The `clamp1()` function with the respective PDG.

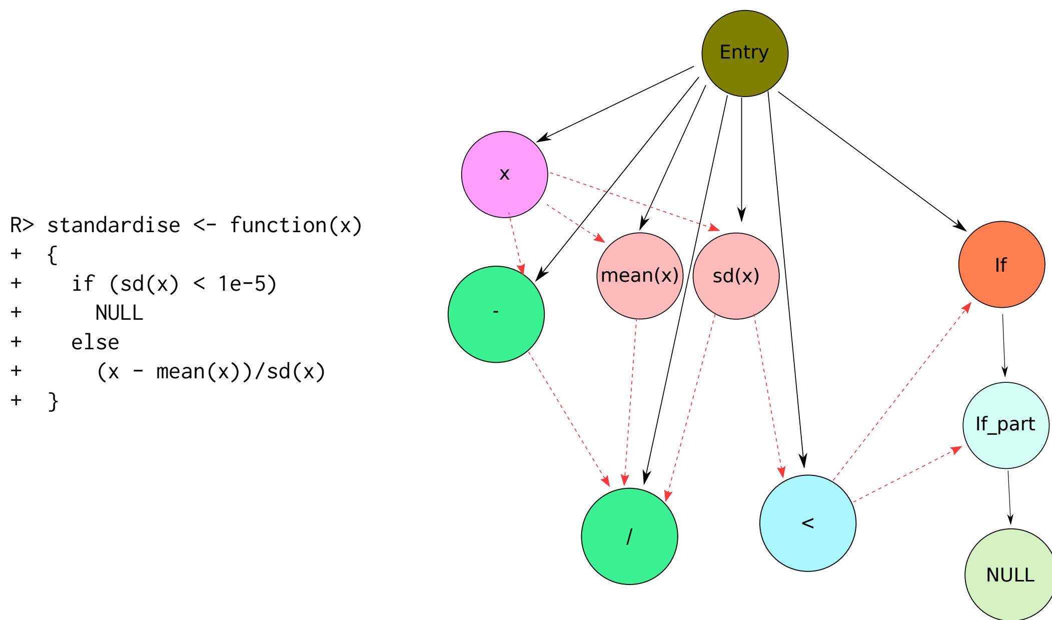

Figure 4: The `standardise()` function with the respective PDG.

1. **Vertex importance degrees.** Firstly, each vertex in F is assigned an importance degree, $\delta_F : V_F \to \mathbb{R}_+$,

$$\delta_F(v) = 0.1 + \sum_{(v,u) \in E_F} \left(1 + 0.1 \mathbb{I}_{\xi_F((v,u))=\mathtt{DATA}}\right) \delta_F(u),$$

   where $\mathbb{I}_{\text{cond}}$ is an indicator function with value 1 if cond is true and 0 otherwise. In other words, a vertex v with outdegree equal to 0 has importance $\delta(v) = 0.1$. Otherwise, its importance degree is set to $\delta(v) = 0.1$ plus the sum of importances of its outgoing control-dependent neighbours plus the sum of importances of its outgoing data-dependent neighbours multiplied by 1.1.

   Note that if F is an acyclic graph, then it has a topological ordering, i.e., an arrangement of the vertices such that every edge is directed from earlier to later in the sequence. In such a case δ_F is well-defined. Otherwise, we shall be computing the importance degrees in the depth-first manner.

Next, the importance degrees are normalised, $\delta_F(v) \mapsto \delta_F(v) / \sum_{u\in V_F} \delta_F(u)$. Tables 2 and 3 give the importance degrees of the vertices in the two graphs studied.

2. **Vertex labels.** Recall from the previous section that each vertex $v \in V_F, u \in V_G$ has been assigned a label, $\zeta_F(v), \zeta_G(u) \in \{0,\ldots,25\}$, based on the type of operation it represents (see Tables 4 and 5).

   In the i-th (out of h in total, here we fix $h = 3$; see Bartoszuk, 2018 for discussion) iteration of the SimilaR algorithm, we assign new labels ζ_F^i, ζ_G^i according to the labels previously considered.

   (a) **Iteration $i = 1$.** In the first iteration, the initial labels, ζ_F, ζ_G, are simply remapped to consecutive integers. This yields ζ_F^1 and ζ_G^1 as given in Tables 4 and 5.

   (b) **Iterations $i = 2$ and $i = 3$.** In subsequent iterations, we seek groups of similar vertices so as to assign them the same label. Two vertices $v, u \in V_F \cup V_G$ are considered *similar* (with no loss in generality, we are assuming $v \in V_F$ and $u \in V_G$ below), whenever they have been assigned the same label in the previous iteration *and* have outgoing neighbours with the same labels. However, for greater flexibility, we allow for the neighbourhoods to differ slightly – unmatched neighbours of lesser importance degrees will not be considered significant. Formally, $v \in V_F$ and $u \in V_G$ are similar, whenever $\zeta_F^{i-1}(v) = \zeta_G^{i-1}(u)$ and:

$$\sum_{(v,w)\in E_F, \zeta_F^{i-1}(w)\notin C(v,u)} \delta_F(w) \le \min\{M_F, M_G\},$$

   where $C(v,u) = \{\zeta_F^{i-1}(w) : (v,w) \in E_F\} \cap \{\zeta_G^{i-1}(w) : (u,w) \in E_G\}$ denotes the *multiset* of common neighbours' vertex labels and M_F and M_G denote the medians of importance degrees of vertices in F and G, respectively.

   The above similarity relation in obviously reflexive and symmetric. When we compute its transitive closure, we get an equivalence relation whose equivalence classes determine sets of vertices that shall obtain identical labels.

   For instance, let $i = 2$ and v be the *Entry* vertex in F and u be the *Entry* vertex in u. We have $\zeta_F^1(v) = \zeta_G^1(u) = 0$. Outgoing neighbours of v have types: 1 (x), 3 (- *(2)*), 3 (*/*), 2 (*min(x)*), 2 (*max(x)*), 4 (<), 3 (- *(1)*) and 5 (*If*). Neighbours of u are labelled as 1 (x), 3 (-), 2 (*mean(x)*), 3 (*/*), 2 (*sd(x)*), 4 (<), 5 (*If*). Hence, v has one unmatched neighbour of type 3 (here: an arithmetic operation). However, as there are 3 such neighbours of v (importances: $0.01, 0.01, 0.06$) and 2 of u (importances: $0.01, 0.02$), we need a rule for determining the left-out importance degree. Here, for greater algorithm's flexibility, we always assume that the unmatched importances are considered in an increasing order. Therefore, we get that $0.01 \le \min\{M_F, M_G\} = \min\{0.04, 0.03\} = 0.03$ and thus v and u are considered similar.

   As another example, let $i = 3$ and v be the x vertex in F and u be the x vertex in G. We have $\zeta_F^2(v) = \zeta_G^2(u) = 1$. Their neighbours are: *- (2)* (importance=0.01; type=6), *min(x)* (importance=0.09; type=2), *max(x)* (importance=0.07; type=2) and - (importance=0.02; type=6), *mean(x)* (importance=0.04; type=2), *sd(x)* (importance=0.10; type=9). The sum of the importance degrees of the unmatched neighbours is now equal to $0.07 + 0.10 > 0.03$, hence, u and v are not considered similar. They are also not in the transitive closure with respect to the other similarities. Therefore, they will be assigned different labels.

3. **Partial similarity degrees.** Let m be the maximal integer label assigned above and $\mathbf{L}_F^i = (\mathbf{L}_{F,1}^i, \ldots, \mathbf{L}_{F,m}^i)$ be a vector of label counts, where $\mathbf{L}_{F,j}^i = |\{v \in V_F : \zeta_F^i(v) = j\}|$, see Table 6. We define $\mathbf{L}_G^i$ in much the same way, see Table 7. We introduce the following "partial" similarity measures of the label sequences – the symmetric:

$$\mu(\mathbf{L}_F^i, \mathbf{L}_G^i) = 1 - \frac{\sum_{k=1}^m |\mathbf{L}_{F,k}^i - \mathbf{L}_{G,k}^i|}{\sum_{k=1}^m \mathbf{L}_{F,k}^i + \sum_{k=1}^m \mathbf{L}_{G,k}^i}$$

   and its nonsymmetric version:

$$\tilde{\mu}(\mathbf{L}_F^i, \mathbf{L}_G^i) = \frac{\sum_{k=1}^m \min(\mathbf{L}_{F,k}^i, \mathbf{L}_{G,k}^i)}{\sum_{k=1}^m \mathbf{L}_{F,k}^i}.$$

   The partial similarities for $i = 1, 2, 3$ are given in Table 8.

4. **Final similarity degrees.** The overall similarity degree is defined as the arithmetic mean of the $h = 3$ partial similarities (reported in Table 8):

$$\mu(F,G) = \frac{1}{h}\sum_{i=1}^{h} \mu(\mathbf{L}_F^i, \mathbf{L}_G^i) = \frac{0.95 + 0.86 + 0.57}{3} = 0.79, \tag{1}$$

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
$\mathbf{L}_{F,j}^{1}$	1	1	2	3	1	1	1	1	0	0	0	0	0
$\mathbf{L}_{F,j}^{2}$	1	1	2	1	1	1	2	1	1	0	0	0	0
$\mathbf{L}_{F,j}^{3}$	1	1	1	1	2	1	2	1	1	0	0	0	0

Table 6: `clamp1()`: Label counts in each iteration.

j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
$\mathbf{L}_{G,j}^{1}$	1	1	2	2	1	1	1	1	0	0	0	0	0
$\mathbf{L}_{G,j}^{2}$	1	1	1	1	1	1	2	1	0	1	0	0	0
$\mathbf{L}_{G,j}^{3}$	1	1	1	1	2	0	0	0	0	1	1	1	1

Table 7: `standardise()`: Label counts in each iteration.

i	$\mu(\mathbf{L}_F^i, \mathbf{L}_G^i)$	$\tilde{\mu}(\mathbf{L}_F^i, \mathbf{L}_G^i)$	$\tilde{\mu}(\mathbf{L}_G^i, \mathbf{L}_F^i)$
1	0.95	0.91	1.00
2	0.86	0.82	0.90
3	0.57	0.55	0.60
Final	0.79	0.76	0.83

Table 8: Similarity measures in each iteration.

We obtain its nonsymmetric versions in the same way:

$$\begin{aligned} \tilde{\mu}(F,G) &= \frac{1}{h}\sum_{i=1}^{h}\tilde{\mu}(\mathbf{L}_F^i,\mathbf{L}_G^i) = \frac{0.91+0.82+0.55}{3} = 0.76, \\ \tilde{\mu}(G,F) &= \frac{1}{h}\sum_{i=1}^{h}\tilde{\mu}(\mathbf{L}_G^i,\mathbf{L}_F^i) = \frac{1.00+0.90+0.60}{3} = 0.83. \end{aligned} \tag{2}$$

Having discussed the algorithms behind the **SimilaR** package, let us proceed with the description of its user interface.

Illustrative examples

The **SimilaR** package can be downloaded from CRAN and installed on the local system via a call to:

```
R> install.packages("SimilaR")
```

Here we are working with version 1.0.8 of the package.

Once the package is loaded and its namespace is attached by calling:

```
R> library("SimilaR")
```

two functions are made available to a user. `SimilaR_fromTwoFunctions()` is responsible for assessing the similarity between a pair of function objects (R is a functional language, hence assuming that functions constitute basic units of code seem natural). Moreover, `SimilaR_fromDirectory()`, which we shall use in the next section, is a conveniently vectorised version of the former, performing the comparison of all the scripts in a given directory.

Let us evaluate the similarity between the $F =$ `clamp1()` and $G =$ `standardise()` functions defined above:

```
R> SimilaR_fromTwoFunctions(clamp1, standardise) # aggregation="sym"

   name1       name2   SimilaR decision
1 clamp1 standardise 0.7954545        1
```

Here, SimilaR denotes the (symmetric) measure $\mu(F,G) \in [0,1]$ as given by Eq. (1). decision uses a built-in classifier model to assess whether such a similarity degree is considered significant (1) or not (0).

To obtain the non-symmetric measures, $\tilde{\mu}(F,G)$ and $\tilde{\mu}(G,F)$ (see Eq. (2)), we pass aggregation="both" as an argument:

```
R> SimilaR_fromTwoFunctions(clamp1, standardise, aggregation="both")

   name1       name2 SimilaR12 SimilaR21 decision
1 clamp1 standardise 0.7575758 0.8333333        1
```

Example: `clamp2()`**.** Let us recall the source code of the clamp1() function:

```
R> clamp1

function(x)
{
  if (max(x)-min(x) < 1e-5)
    NULL
  else
    (x - min(x))/(max(x)-min(x))
}
```

By applying numerous transformations described above, we may arrive at its following version:

```
R> library("magrittr")
R> clamp2 <- function(y)
+  {
+    longName <- y                   # variable duplication
+    longName2 <- min
+    z <- { sum(longName**2) }       # dead code
+    min_y <- longName %>% longName2 # forward-pipe
+    max_y <- y %>% max
+    max_y_min_y <- max_y-min_y # memoization
+    if(!(max_y_min_y >= 1e-5)) # canonicalization of the if statement
+    {
+      return(NULL)
+    }
+    ((y - min_y)/max_y_min_y) # tail call to return removed
+  }
```

SimilaR correctly identifies the two functions as equivalent:

```
R> SimilaR_fromTwoFunctions(clamp1, clamp2, aggregation="both")

   name1  name2 SimilaR12 SimilaR21 decision
1 clamp1 clamp2         1         1        1
```

Example: A vectorised version of `clamp1()`**.** Let us now consider two different vectorised versions of the clamp1() function for list-type inputs. The first one is based on a call to lapply(), which takes care of applying a given anonymous function on each list member:

```
R> clamp1_vectorised1 <- function(x) {
+      x %>% lapply(function(y) {
+          if (max(y)-min(y) < 1e-5) {
+              {{{{NULL}}}}
+          } else {
+              {{{{(y - min(y))/(max(y)-min(y))}}}}
+          }
+      })
+  }
```

The second function unwinds the Map-like construct, adding a for-loop instead. It also relies on some domain knowledge, namely, that a new list is pre-allocated with NULLs.

```
R> clamp1_vectorised2 <- function(x) {
+     n <- length(x)
+     res <- vector("list", n) # NULLs
+     for (i in 1:n) { # assumed n>0
+         m <- min(x[[i]])
+         mm <- max(x[[i]])-m
+         if (mm >= 1e-5)
+             res[[i]] <- (x[[i]] - m)/mm
+     }
+     return(res)
+ }
```

Note that the function only works for non-empty input lists.

The pairwise comparison yields:

```
R> SimilaR_fromTwoFunctions(clamp1_vectorised1, clamp1_vectorised2,
+     aggregation="both")

                name1              name2 SimilaR12 SimilaR21 decision
1 clamp1_vectorised1 clamp1_vectorised2 0.7833333 0.9215686        1
```

Which indicates a significant degree of similarity, which is indeed the case.

A case study

In the previous section we illustrated that **SimilaR** is easily able to identify the code chunks that can be transformed *onto* each other. Now we shall demonstrate its usefulness in a real-world scenario: let us compare the code-base of two R packages: **nortest** (Gross and Ligges, 2015) and **DescTools** (Signorell et al., 2020). The former implements five significance tests for normality, while the latter advertises itself as

> *A collection of miscellaneous basic statistic functions and convenience wrappers for efficiently describing data. [...] Many of the included functions can be found scattered in other packages and other sources written partly by Titans of R. The reason for collecting them here, was primarily to have them consolidated in ONE instead of dozens of packages (which themselves might depend on other packages which are not needed at all), and to provide a common and consistent interface as far as function and arguments naming,* `NA` *handling, recycling rules, etc. are concerned. [...]*
> *(***DescTools** *package* `DESCRIPTION`*; Signorell et al., 2020)*

1. Set-up. First we attach the required packages and set up the directory where we shall store the data that we are going to feed the algorithm with at a later stage.

```
R> library("SimilaR")
R> dir_output <- tempfile("dir")
R> dir.create(dir_output)
```

2. Generate code-base. **SimilaR** does not offer direct support for comparing two R packages. Therefore, below we export each package's code-base to a single source file, getting rid of non-function objects:

```
R> for (pkg in c("DescTools", "nortest"))
+ {
+    library(pkg, character.only=TRUE)  # attach the package
+    env <- as.environment(paste0("package:", pkg)) # package's environment
+    fun_names <- ls(envir=env)  # list of all exported objects
+    file_text <- character(0) # the to-be code-base (1 function == 1 string)
+    for (fun in fun_names)
+    {
+       f <- get(fun, env)   # get function object
+       if (!is.function(f)) next
+
+       f_char <-  paste0(deparse(f), collapse="\n") # extract source code
```

```
+       file_text[length(file_text)+1] <- sprintf("`%s`<-%s", fun, f_char)
+     }
+     file_name <- file.path(dir_output, paste0(pkg, ".R"))
+     writeLines(file_text, file_name) # write source file
+     cat(sprintf("%s: %d functions processed.\n", pkg, length(file_text)))
+   }

DescTools: 549 functions processed.
nortest: 5 functions processed.
```

Here the list of the objects exported by both packages is determined by querying their corresponding `package:DescTools` and `package:nortest` environments. Moreover, a call to `deparse()` on a function object gives a plain-text representation of its source code.

3. Run the algorithm. Now we ask the algorithm to fetch the two source files in the output directory and execute all the pairwise comparisons between the functions defined therein.

```
R> time0 <- Sys.time()
R> results <- SimilaR_fromDirectory(dir_output,
+      fileTypes="file", aggregation="both")
R> print(Sys.time()-time0)

Time difference of 15.41459 secs
```

The above gives the time to execute all the 2745 pairwise comparisons on an Intel Core i7 laptop with 16GB RAM, running the GNU/Linux 4.19.30-041930-generic SMP x86_64 kernel.

4. Report results. Let us inspect the top 10 results returned by the algorithm (in terms of overall similarity).

```
R> print(head(results, 10))
```

For greater readability, the results are reported in Table 9.

Name1 (DescTools)	Name2 (nortest)	SimilaR12	SimilaR21	Decision
`CramerVonMisesTest()`	`cvm.test()`	1.0000000	1.0000000	1
`LillieTest()`	`lillie.test()`	1.0000000	1.0000000	1
`PearsonTest()`	`pearson.test()`	1.0000000	1.0000000	1
`ShapiroFranciaTest()`	`sf.test()`	1.0000000	1.0000000	1
`CramerVonMisesTest()`	`ad.test()`	0.9451477	0.9105691	1
`CramerVonMisesTest()`	`lillie.test()`	0.7299578	0.5339506	0
`LillieTest()`	`cvm.test()`	0.5339506	0.7299578	0
`LillieTest()`	`ad.test()`	0.5339506	0.7032520	0
`Eps()`	`sf.test()`	0.5333333	0.6315789	0
`ShapiroFranciaTest()`	`ad.test()`	0.7719298	0.3577236	0

Table 9: Similarity report (the top 10 results) for the comparison between the code-base of the **DescTools** and **nortest** packages.

Discussion. We observe that 5 function pairs were marked as similar (`Decision` = 1). The top 4 results accurately indicate the corresponding normality tests from the two packages – their sources are identical.

However, the 5th result is a *false positive*: `CramerVonMisesTest()` is reported as similar to `ad.test()`, which implements the Anderson–Darling normality test. Let us "visually" compare their sources:

```
CramerVonMisesTest <- function(x) {
    DNAME <- deparse(substitute(x))
    x <- sort(x[complete.cases(x)])
    n <- length(x)
    if (n < 8)
```

```
        stop("sample size must be greater than 7")
    p <- pnorm((x - mean(x))/sd(x))
    W~<- (1/(12 * n) + sum((p - (2 * seq(1:n) - 1)/(2 * n))^2))
    WW <- (1 + 0.5/n) * W
    if (WW < 0.0275) {
        pval <- 1 - exp(-13.953 + 775.5 * WW - 12542.61 * WW^2)
    }
    else if (WW < 0.051) {
        pval <- 1 - exp(-5.903 + 179.546 * WW - 1515.29 * WW^2)
    }
    else if (WW < 0.092) {
        pval <- exp(0.886 - 31.62 * WW + 10.897 * WW^2)
    }
    else if (WW < 1.1) {
        pval <- exp(1.111 - 34.242 * WW + 12.832 * WW^2)
    }
    else {
        warning("p-value is smaller than 7.37e-10,
            cannot be computed more accurately")
        pval <- 7.37e-10
    }
    RVAL <- list(statistic = c(W = W), p.value = pval,
        method = "Cramer-von Mises normality test",
        data.name = DNAME)
    class(RVAL) <- "htest"
    return(RVAL)
}

ad.test <- function(x) {
    DNAME <- deparse(substitute(x))
    x <- sort(x[complete.cases(x)])
    n <- length(x)
    if (n < 8)
        stop("sample size must be greater than 7")
    logp1 <- pnorm((x - mean(x))/sd(x), log.p = TRUE)
    logp2 <- pnorm(-(x - mean(x))/sd(x), log.p = TRUE)
    h <- (2 * seq(1:n) - 1) * (logp1 + rev(logp2))
    A~<- -n - mean(h)
    AA <- (1 + 0.75/n + 2.25/n^2) * A
    if (AA < 0.2) {
        pval <- 1 - exp(-13.436 + 101.14 * AA - 223.73 * AA^2)
    }
    else if (AA < 0.34) {
        pval <- 1 - exp(-8.318 + 42.796 * AA - 59.938 * AA^2)
    }
    else if (AA < 0.6) {
        pval <- exp(0.9177 - 4.279 * AA - 1.38 * AA^2)
    }
    else if (AA < 10) {
        pval <- exp(1.2937 - 5.709 * AA + 0.0186 * AA^2)
    }
    else pval <- 3.7e-24
    RVAL <- list(statistic = c(A = A), p.value = pval,
        method = "Anderson-Darling normality test",
        data.name = DNAME)
    class(RVAL) <- "htest"
    return(RVAL)
}
```

Basically, the main difference between the two functions is in the numeric constants used, which we know is a kind of detail purposefully ignored by **SimilaR**. Indeed, knowing that the Anderson–Darling test is a generalisation of the Cramér–von Mises test (both of them are based on the L_2-distance between the empirical and true cumulative distribution function, the former is a weighted version of the latter), makes the reported decision concerning the *similarity* judgement quite justified.

Interestingly, **DescTools** does provide the `AndersonDarlingTest` function, but this is a version of

the goodness-of-fit measure to test against *any* probability distribution provided. In other words, a c.d.f. of a normal distribution is not hard-coded in its source, and thus is significantly different from the code of `ad.test()`.

It is also worth noting that there are no false positives in terms of statistical tool types – all the functions deal with some form of goodness-of-fit testing and we recall that **DescTools** defines ca. 550 functions in total.

Discussion

We have introduced an algorithm to quantify the similarity between a pair of R source code chunks. The method is based on carefully prepared *Program Dependence Graphs*, which assure that semantically equivalent code pieces are represented in the same manner even if they are written in much different ways. This makes the algorithm robust with respect to the most typical *attacks*. In a few illustrative examples, we have demonstrated typical code alterations that the algorithm is invariant to, for instance, aliasing of variables, changing the order of independent code lines, unwinding nested function calls, etc.

In the presented case study we have analysed the similarities between the **DescTools** and **nortest** packages. Recall that most of the cloned function pairs are correctly identified, proving the practical usefulness of the **SimilaR** package. The reported above-threshold similarity between `CramerVonMisesTest()` and `ad.test()` is – strictly speaking – a false positive, nevertheless our tool correctly indicates that the two functions have been implemented in much the same way. This might serve as a hint to package developers that the two tests could be refactored so as to rely on a single internal function – de-duplication is among the most popular ways to increase the broadly conceived quality of software code.

On the other hand, the algorithm failed to match the very much-different (implementation-wise) `AndersonDarlingTest()` (generic distribution) with its specific version of `ad.test()` (normal distribution family). However, comparisons of such a kind, in order to be successful, would perhaps require the use of an extensive knowledge-base and are of course beyond the scope of our tool.

Finally, let us note that due to the use of a new polynomial-time algorithm, assessing the similarity of two Program Dependence Graphs is relatively fast. This makes **SimilaR** appropriate for mining software repositories even of quite considerable sizes. However, some pre-filtering of function pairs (e.g., based on cluster analysis) to avoid performing all the pairwise comparisons would make the system even more efficient and scalable.

Future versions of the **SimilaR** package will be equipped with standalone routines aiming at improving the quality and efficiency of R code, such as detecting dead or repeated code, measuring cyclomatic complexity, checking if the program is well structured, etc.

Acknowledgements. The authors would like to thank the Reviewers for valuable feedback that helped improve this manuscript.

Bibliography

H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996. [p368]

A. T. Ali, H. M. Abdulla, and V. Snasel. Overview and comparison of plagiarism detection tools. In *Proceedings of the Dateso 2011: Annual International Workshop on Databases, Texts, Specifications and Objects*, Dateso '11, pages 161–172, 2011. [p367]

P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2013. [p367]

S. M. Bache and H. Wickham. *magrittr: A Forward-Pipe Operator for R*, 2014. URL https://CRAN.R-project.org/package=magrittr. R package version 1.5. [p368]

M. Bartoszuk. *A Source Code Similarity Assessment System for Functional Programming Languages Based on Machine Learning and Data Aggregation Methods*. PhD thesis, Warsaw University of Technology, Warsaw, Poland, 2018. [p368, 370, 374, 376]

I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–378, 1998. [p367]

R. Becker, J. Chambers, and A. Wilks. *The New S Language*. Chapman & Hall, 1998. [p368]

J. Chambers. *Programming with Data*. Springer, 1998. [p368]

J. Chambers. *Software for Data Analysis. Programming with R*. Springer-Verlag, 2008. [p368]

J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987. URL https://doi.org/10.1145/24039.24041. [p368]

D. Fu, Y. Xu, H. Yu, and B. Yang. WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection. *Scientific Programming*, 2017:7809047, 2017. [p368]

M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 321–330, 2008. URL https://doi.org/10.1145/1368088.1368132. [p368]

A. Ghosh and Y. Lee. An Empirical Study of a Hybrid Code Clone Detection Approach on Java Byte Code. *GSTF Journal on Computing*, 5(2):34–45, 2018. [p368]

S. Grant, J. R. Cordy, and D. B. Skillicorn. Using topic models to support software maintenance. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 403–408, 2012. URL https://doi.org/10.1109/CSMR.2012.51. [p367]

J. Gross and U. Ligges. *nortest: Tests for Normality*, 2015. URL https://cran.r-project.org/package=nortest. R package version 1.0-4. [p379]

J. Hage, P. Rademaker, and N. van Vugt. Plagiarism Detection for Java: A Tool Comparison. In *Computer Science Education Research Conference*, CSERC '11, pages 33–46, 2011. [p367]

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. [p372]

S. Horwitz and T. Reps. Efficient comparison of program slices. *Acta Informatica*, 28(8):713–732, 1991. URL https://doi.org/10.1007/BF01261653. [p368]

B. Hummel, E. Juergens, and D. Steidl. Index-based model clone detection. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 21–27, 2011. URL https://doi.org/10.1145/1985404.1985409. [p367]

L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, 2007. URL https://doi.org/10.1109/ICSE.2007.30. [p368]

J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering – Volume 1*, CASCON '93, pages 171–183, 1993. [p367]

T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002. URL https://doi.org/10.1109/TSE.2002.1019480. [p367]

R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, 2001. [p368]

J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 301–307, 2001. [p368]

Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006. URL https://doi.org/10.1109/TSE.2006.28. [p367]

E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining concepts from code with probabilistic topic models. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 461–464, 2007. ISBN 978-1-59593-882-4. URL https://doi.org/10.1145/1321631.1321709. [p367]

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, 2006. URL https://doi.org/10.1145/1150402.1150522. [p368]

S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source code retrieval for bug localization using latent Dirichlet allocation. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, WCRE '08, pages 155–164, 2008. ISBN 978-0-7695-3429-9. URL https://doi.org/10.1109/WCRE.2008.33. [p367]

M. Luperto and F. Amigoni. Predicting the global structure of indoor environments: A constructive machine learning approach. *Autonomous Robots*, 43(4):813–835, 2019. [p373]

U. Manber. Finding similar files in a large file system. In *USENIX Winter 1994 Technical Conference*, pages 1–10, 1994. [p367]

P. Mapar. Machine learning for enzyme promiscuity. Master's thesis, Aalto University, Finland, 2018. [p373]

A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 107–114, 2001. [p367]

V. T. Martins, D. Fonte, P. R. Henriques, and D. da Cruz. Plagiarism Detection: A Tool Survey and Comparison. In M. J. V. Pereira et al., editors, *3rd Symposium on Languages, Applications and Technologies*, OpenAccess Series in Informatics (OASIcs), pages 143–158, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/OASIcs.SLATE.2014.143. [p367]

G. Maskeri, S. Sarkar, and K. Heafield. Mining business topics in source code using latent Dirichlet allocation. In *Proceedings of the 1st India Software Engineering Conference*, ISEC '08, pages 113–120, 2008. ISBN 978-1-59593-917-3. URL https://doi.org/10.1145/1342211.1342234. [p367]

J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *1996 Proceedings of International Conference on Software Maintenance*, pages 244–253, 1996. URL https://doi.org/10.1109/ICSM.1996.565012. [p368]

P. W. McBurney, C. Liu, C. McMillan, and T. Weninger. Improving topic model source code summarization. In *Proc. 22nd International Conference on Program Comprehension*, ICPC 2014, pages 291–294, 2014. ISBN 978-1-4503-2879-1. URL https://doi.org/10.1145/2597008.2597793. [p367]

M. Misic, Z. Siustran, and J. Protic. A comparison of software tools for plagiarism detection in programming assignments. *International Journal of Engineering Education*, 32(2):738–748, 2016. [p367, 368]

A. Mohd Noor et al. Programming similarity checking system. *Journal of Telecommunication, Electronic and Computer Engineering*, 9(3–5):89–94, 2017. [p367]

H. Nasirloo and F. Azimzadeh. Semantic code clone detection using abstract memory states and program dependency graphs. In *Proc. 4th International Conference on Web Research (ICWR)*, pages 19–27, 2018. [p368]

J. F. Patenaude, E. Merlo, M. Dagenais, and B. Lague. Extending software quality assessment techniques to Java systems. In *Proceedings Seventh International Workshop on Program Comprehension*, pages 49–56, 1999. URL https://doi.org/10.1109/WPC.1999.777743. [p368]

L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, University of Karlsruhe, Department of Informatics, 2000. [p367, 368]

W. Qu, Y. Jia, and M. Jiang. Pattern mining of cloned codes in software systems. *Information Sciences*, 259:544–554, 2014. URL https://doi.org/10.1016/j.ins.2010.04.022. [p368]

D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013. URL https://doi.org/10.1016/j.infsof.2013.01.008. [p367]

M. Rieger. *Effective clone detection without language barriers*. PhD thesis, University of Bern, Switzerland, 2005. [p367]

C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009. URL https://doi.org/10.1016/j.scico.2009.02.007. [p367]

S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, 2003. URL https://doi.org/10.1145/872757.872770. [p367, 368]

K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proc. USENIX Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association. [p367]

N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler–Lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011. [p373]

A. Signorell et al. *DescTools: Tools for Descriptive Statistics*, 2020. URL https://cran.r-project.org/package=DescTools. R package version 0.99.37. [p379]

J. Silge, J. C. Nash, and S. Graves. Navigating the R Package Universe. *The R Journal*, 10(2):558–563, 2018. URL https://doi.org/10.32614/RJ-2018-058. [p368]

K. Tian, M. Revelle, and D. Poshyvanyk. Using latent Dirichlet allocation for automatic categorization of software. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 163–166, 2009. URL https://doi.org/10.1109/MSR.2009.5069496. [p367]

Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *Ninth Asia-Pacific Software Engineering Conference, 2002.*, pages 327–336, 2002. URL https://doi.org/10.1109/APSEC.2002.1183002. [p367]

M. van der Loo. The stringdist package for approximate string matching. *The R Journal*, 6:111–122, 2014. [p367]

Vandana. A comparative study of plagiarism detection software. In *2018 5th International Symposium on Emerging Trends and Technologies in Libraries and Information Services*, pages 344–347. IEEE, 2018. URL https://doi.org/10.1109/ETTLIS.2018.8485271. [p368]

W. Venables and B. Ripley. *S Programming*. Springer, 2000. [p368]

I. Wegener. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Springer, 2005. [p373]

B. Weisfeiler and A. A. Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968. [p373]

H. Wickham. *Advanced R*. Chapman & Hall/CRC, 2014. [p368]

H. Wickham and G. Grolemund. *R for Data Science*. O'Reilly, 2017. [p368]

M. J. Wise. Detection of similarities in student programs: YAP'ing may be preferable to Plague'ing. *ACM SIGCSE Bulletin*, 24(1):268–271, 1992. URL https://doi.org/10.1145/135250.134564. [p367]

Maciej Bartoszuk
Faculty of Mathematics and Information Science,
Warsaw University of Technology
ul. Koszykowa 75, 00-662 Warsaw, Poland
https://bartoszuk.rexamine.com
https://orcid.org/0000-0001-6088-8273
m.bartoszuk@mini.pw.edu.pl

Marek Gagolewski
School of Information Technology,
Deakin University
Geelong, VIC, Australia
and
Faculty of Mathematics and Information Science,
Warsaw University of Technology
ul. Koszykowa 75, 00-662 Warsaw, Poland
https://www.gagolewski.com
https://orcid.org/0000-0003-0637-6028
m.gagolewski@deakin.edu.au