Detecting Similar Programs via The Weisfeiler-Leman Graph Kernel

Wenchao Li[1], Hassen Saidi[1], Huascar Sanchez[1], Martin Schäf[1](✉), and Pascal Schweitzer[2]

[1] SRI International, 333 Ravenswood Ave, Menlo Park 94025, USA
martin.schaef@sri.com
[2] RWTH Aachen University, Aachen, Germany
http://www.csl.sri.com/people/li/
http://www.csl.sri.com/people/saidi/
https://huascarsanchez.com/
http://www.csl.sri.com/people/schaef/
http://www.lii.rwth-aachen.de/~schweitzer/

Abstract. With the increasing availability of source code on the Internet, many new approaches to retrieve, repair, and reuse code have emerged that rely on the ability to efficiently compute the similarity of two pieces of code. The meaning of similarity, however, heavily depends on the application domain. For predicting API calls, for example, programs can be considered similar if they call a specific set of functions in a similar way, while for automated bug fixing, it is important that similar programs share a similar data-flow.

In this paper, we propose an algorithm to compute program similarity based on the Weisfeiler-Leman graph kernel. Our algorithm is able to operate on different graph-based representations of programs and thus can be applied in different domains. We show the usefulness of our approach in two experiments using data-flow similarity and API-call similarity.

1 Introduction

Over the past few years, we have seen a rapid increase in the amount of source code that is openly available on the Internet. Source code hosting platforms such as GitHub, BitBucket, or SourceForge and social media resources like StackOverflow have changed the way we program. This large amount of machine readable source code also has given rise to several interesting new research directions, such as code prediction [23], discovery of architectural patterns [20], using donor code for program repair [12,29], and more efficient ways to search for code [17,30].

Central to these new approaches is the ability to efficiently find *similar* code snippets in the wild. Unlike in traditional code clone detection, the notion of similarity depends heavily on the application. For automatic program repair, for example, it is important that code shares a similar data-flow, whereas for code

This work is funded in parts by AFRL contract No. FA8750-15-C-0010.

© Springer International Publishing Switzerland 2016
G.M. Kapitsaki and E. Santana de Almeida (Eds.): ICSR 2016, LNCS 9679, pp. 315–330, 2016.
DOI: 10.1007/978-3-319-35122-3_21

prediction, it is often sufficient if the code interacts with a certain API in a similar way. Hence, finding a generic approach to comparing code that can work with different representations and abstractions has the potential to be beneficial in a variety of fields.

To address this problem, we propose a new algorithm to compute a program similarity score based on a technique from graph isomorphism testing. Here, we use the term *program* as a shorthand for any piece of code, like a full program, isolated classes or methods, or just snippets.

Our algorithm consists of two parts. First, it turns a program into a labeled graph. Second, it computes a Weisfeiler-Leman kernel for this graph and compares it against the precomputed kernels of the graphs of other programs to identify the most similar ones.

The algorithm itself is agnostic to the graph representation of the program or the programming language and can be used with different graphs. To illustrate the usefulness of this approach, we introduce two graph representations of Java programs, a simplified inter-procedural data-flow graph (IDFG), and an API-call graph (ACG), and evaluate how these graphs can be used in combination with our algorithm to identify similar programs. The IDFG is a simplified version of the actual data-flow graph of a Java program and suitable to find programs that are algorithmically similar, while the ACG is a stripped-down version of an inter-procedural control-flow graph that only contains calls to a given API, which is suitable to find examples of API usage.

To evaluate the ability of our approach to identify similar programs, we conduct two experiments. For the first experiment, we choose a subset of the Google CodeJam[1] corpus as a benchmark. The corpus is a set of 4 algorithmic problems, each with hundreds of solutions given as small Java programs (in total 1,280 programs). Our goal is to show that, when picking any of these Java programs, our approach for finding similarities using the IDFG identifies similar programs that are in fact solutions to the same problem.

The programs in CodeJam are very algorithmic and make only limited use of API calls (e.g., for printing). Hence, this corpus is unfortunately not suitable to evaluate our approach in combination with the second graph representation, the ACGs. Thus, we perform a second experiment where we use the Apache `commons-lang` project as a benchmark. We compare the similarity between all pairs of methods in this application and evaluate manually if the reported similarities indicate similar API usage patterns.

Roadmap. In the following Section, we introduce the Weisfeiler-Leman algorithm to compute graph kernels for labeled graphs. In Sect. 3, we explain how we use these graph kernels to compute a similarity score between graphs. In Sect. 4, we introduce the two graph representations of Java programs (IDFG and ACG) that we use in our experiments. We evaluate our approach in Sect. 5, discuss the related work in Sect. 6, and propose future directions in Sect. 7.

[1] https://code.google.com/codejam.

2 Preliminaries

Our approach to measure the similarity between programs is based on a standard routing from graph isomorphism testing which we introduce in this section. More specifically, we use the 1-dimensional Weisfeiler-Leman algorithm, often also referred to as color refinement or naïve vertex classification. The procedure is for example employed in the currently fastest practical isomorphism solvers (such as `Nauty` and `Traces` [18], `Bliss` [11] and `saucy` [4]).

The algorithm repeatedly recolors the vertices of its inputs graphs. Starting with an initial coloring of the vertices which distinguishes them by their degree the algorithm proceeds in rounds. In each round the new color of a vertex encodes the previous color as well as the multiset of the colors of the neighbors. The k-dimensional variant, which we will not require in this work, colors k-tuples of vertices and can solve isomorphism on quite general graph classes (see [8]).

Next, we describe the one-dimensional variant more formally. If $G = (V, E, \chi_0)$ is a vertex colored graph, where χ_0 is a vertex coloring, we recursively define χ_{i+1} to be the coloring given by

$$\chi_{i+1}(v) := \big(\chi_i(v), \{\!\{\chi_{i+1}(v') \mid \{v, v'\} \in E\}\!\}\big).$$

Here we use "$\{\!\{$" and "$\}\!\}$" to indicate multisets.

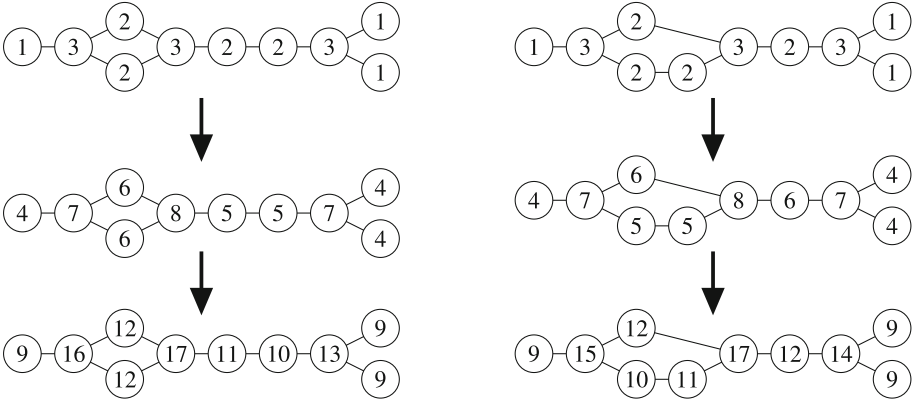
The process leads to an ever finer classification of the vertices. This process stabilizes at some point. In the context of graph isomorphism the histogram of the colors is used to distinguish graphs according to isomorphism.

However, for our purpose, the final colors are excessively descriptive, in the sense that they actually encode too much information. Indeed, by a result from [1], for almost all graph, each final color already encodes the isomorphism type of the graph.

Circumventing this problem we adopt the technique from [27] to only execute the algorithm for a few rounds and exploit the histograms of the colors that appear during the execution of the algorithm. In [27] these histograms are used to design a graph kernel which can then be applied in a machine learning fashion to perform for example graph classification. Said kernel captures similar information to other kernels that count subgraphs (see for example [9] or [28]). However, the Weisfeiler-Leman Kernel can be far more efficiently computed.

It is well known that the exact information captured by i iterations of naïve vertex classification can be precisely expressed in a certain type of logic (see [19]). However, it is difficult to grasp what the result means in terms of graph theoretic properties of the input graphs. While in a regular graph no information is generated at all, since all vertices have the same color in all iterations, in non-regular graphs typically the isomorphism type of a small neighborhood of a vertex is determined.

Concerning running time it is possible to perform h-iterations of the algorithm in $O(hm)$ time, where m is the number of edges of the input graph. To achieve such a running time, the labels have to be compressed to prevent label names from becoming excessively long. There are two options to do this, one



histogram	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
G_L	3	4	3	3	2	2	2	1	3	1	1	2	1	0	0	1	1
G_R	3	4	3	3	2	2	2	1	3	1	1	2	0	1	1	0	1

Fig. 1. The figure shows a left graph G_L and a right graph G_R to which the 1-dimensional Weisfeiler-Leman is being applied for 2 iterations. Here labels are hashed to smaller values. The renaming is as follows $4 := (1, \{\!\{3\}\!\}), 5 := (2, \{\!\{2, 3\}\!\}), 6 := (2, \{\!\{3, 3\}\!\}), 7 := (3, \{\!\{1, 2, 2\}\!\}), 8 := (3, \{\!\{2, 2, 2\}\!\}), 9 := (4, \{\!\{7\}\!\})$ and so on. The figure also shows the histograms of labels that appear in the two graphs highlighting their similarity.

employs techniques such as bucket sort, while the other one simply uses a hash function to compress the labels (see [27]). We adopt the latter approach and used the built-in hash function for strings in our implementation.

In our intended application, we benefit from the fact that the algorithm can take vertex-labeled graphs as input. Thus it is easy to introduce labeled nodes into the algorithm. In Sect. 4 we show how Java types (for IDFG) or method signatures (for the ACG) can be used as initial coloring χ_0 in the definition of the algorithm.

3 Similarities

Our goal is to design a similarity score $S(P, P')$ between programs P and P' with the following properties.

- All programs are 100 % similar to themselves, i.e. $S(P, P) = 1$.
- The score is symmetric, i.e. $S(P, P') = S(P', P)$.
- The score is normalized to a percentage number, i.e. $0 \leq S(P, P') \leq 1$.

A percentage score also has the advantage of being more easily interpreted by the user. We remark that an asymmetric score might be desired in some setting.

For example, the user may be interested in finding a similar program that is also of similar size.

Recall that the Weisfeiler-Leman kernel with h iterations for graphs G and G' is defined as follows [16]:

$$K_{WL}^{(h)}(G, G') = w_0 K(G_0, G'_0) + w_1 K(G_1, G'_1) + \ldots + w_h K(G_h, G'_h) \quad (1)$$

where the G_is and G'_is are the graphs produced by successive recoloring of the original labeled graphs G_0 and G'_0 (as shown in Fig. 1), and $K(G_i, G'_i)$ is a graph kernel for graphs G_i and G'_i. $K_{WL}^{(h)}(G, G')$ is then constructed as a positive linear combination of the $K(G_i, G'_i)$s using some positive weights w_is.

In each iteration of the Weisfeiler-Lehman algorithm, a histogram is produced which encodes certain structural information of the graph, as shown in Fig. 1. Treating these histograms as vectors, a natural candidate for K is the scalar product of two vectors. However, this can be problematic when the graph sizes are very different. Consider the scenario of two graphs G and G' where $|V| \ll |V'|$, and all the nodes in G' have an identical label l which exists somewhere in G. We presume G and G' correspond to either the API call graph or the inter-procedural data-flow graph generated from programs P and P' respectively. Observe that it is easy to have $K(G, G) < K(G, G')$. As a result, P' might be reported as a more similar program to P than P itself. Hence, we apply standard normalization using the lengths of the two vectors and use the angle between the vectors as our similarity measure.

Let $v(G_i)$ present the coloring vector (histogram) produced at the ith iteration of the Weisfeiler-Leman algorithm. Then $K(G_i, G'_i) = v(G_i) \cdot v(G'_i)/\|(v(G_i))\| \; \|(v(G'_i))\|$. Since we are interested in a percentage score, and $0 \leq K(G_i, G'_i) \leq 1$ for each i, the weights w_is can be chosen appropriately to make $0 \leq K_{WL}^{(h)}(G, G') \leq 1$. In our experiments, we simply choose a uniform weight.

4 Graph-Based Program Representations

The ability of our algorithm to detect program similarity strongly depends on the graph representation of programs that we use when computing the graph kernels. Choosing a graph representation for programs is a trade off between precision and the ability to identify programs that serve a similar purpose but are structurally different. Using the precise control-flow graph of a program, for example, would be an efficient way to identify exact code clones but would reduce the chance to identify programs that are semantically similar but syntactically different.

In this paper, we search a balance between precision and flexibility by proposing two graph representations. We call our first graph *API Call Graph* (ACG) which is a stripped-down inter-procedural control-flow graph that only shows calls to a specific set of APIs. Our second graph is a (simplified) *inter-procedural data-flow graph* (IDFG) which tracks the flow of data between memory locations from a given program entry point.

```
1  public void reverseFile(String infile, String outfile)
       throws IOException {
2    String data = readFile(infile);
3    data = new StringBuilder(data).reverse().toString();
4    writeFile(outfile, data);
5  }
6
7  private String readFile(String fname) throws IOException
       {
8    byte[] encoded = Files.readAllBytes(Paths.get(fname));
9    return new String(encoded, Charset.defaultCharset());
10 }
11
12 private void writeFile(String fname, String data) {
13   try (Writer out = new BufferedWriter(
14       new OutputStreamWriter(new FileOutputStream(fname),
15           Charset.defaultCharset()));) {
16     out.write(data);
17   } catch (Exception e) {
18     e.printStackTrace();
19   }
20 }
```

Fig. 2. Running example for our graph representation of programs. The method `reverseFile` is the entry point. It takes two file names as input. It reads the content from the first file to a String using the method `readFile`, reverses the String, and writes out the reverted String using `writeFile`.

Both graphs provide a certain level of abstraction. In the following we discuss the advantages and disadvantages of our design choices along the running example program in Fig. 2. This program has a (public) method `reverseFile` that takes the names of two files as input, calls `readFile` to read the contents of the first file into a string, reverses this string, and subsequently calls `writeFile` to write the resulting string to a file.

We now construct the graph representations for this program using Soot [31]. Soot first translates the program into an intermediate format called Jimple which provides us with a canonical form for expressions. For the graph construction, we extend Soot's flow analysis to collect the node and edges of our graph per method, where we keep placeholder nodes for method calls. Then, in a final step, we substitute the placeholder nodes for method calls by their corresponding graphs. We will discuss details and practical issues of the graph construction in more detail later on in the evaluation.

4.1 API Call Graph

The first graph that we construct is called API Call Graph which represents the order in which procedures of some given APIs can be called from a given entry point.

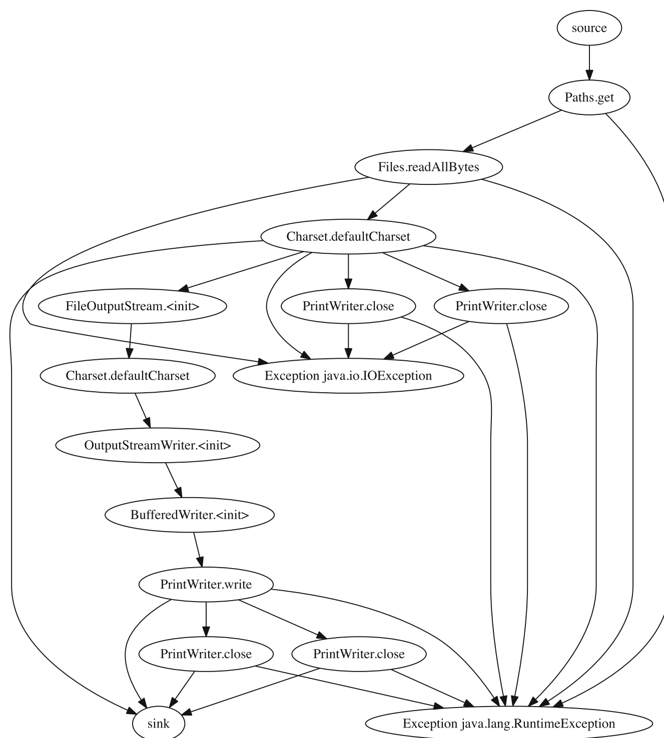


Fig. 3. ACG for our example from Fig. 2. The ACG approximates the possible sequences of calls to methods in `java.io.*` and `java.nio.*`. The labels are simplified for readability and usually also encode the types of parameters and return values.

The motivation for using an ACG for computing program similarity is our hypothesis that, in Java, programmers often achieve their goals by using external APIs. Programs that use the same API calls in the same order should have a similar objective, regardless of the statements in between. A programmer using our approach to find similar code might thus be interested in finding any code that shares the usage pattern of a specific subset of APIs.

For the construction of the ACG, we pick an entry point (e.g., `reverseFile` in our example) and the APIs that we are interested in (e.g., in our example we are interested `java.io.*` and `java.nio.*`). Then we follow the control-flow of the program starting from the entry point. Each time we encounter a call to a method that is declared in an API that we are interested in, we create a new node for this call. Figure 3 shows the ACG for our example from Fig. 2.

The ACG always has a unique source node but can have multiple sink nodes. One unique sink node always represents the normal termination of a procedure, other sink nodes may exist for exceptional returns. In Fig. 2 we have exceptional sinks for `RuntimeException` and `IOException`.

Each time we encounter a call to a method that is not part of an API that we are interested in, but for which we have code available, we inline the method

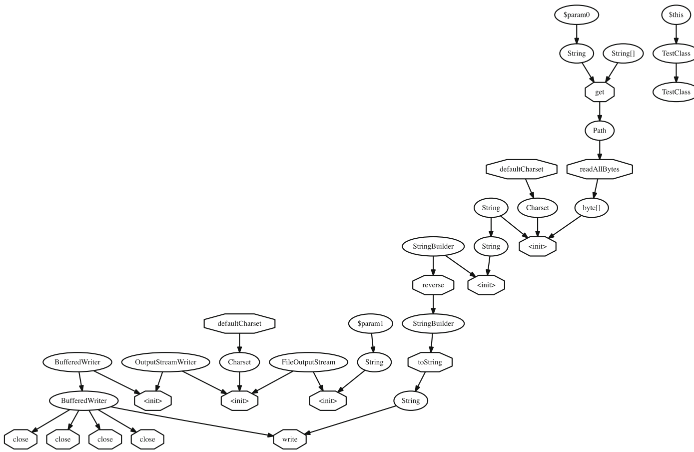


Fig. 4. IDFG for our example from Fig. 2. Vertices are labeled with variable types, edges represent data-flow. Octagon shaped vertices represent method calls. The actual operation are omitted from the edges since our algorithm ignores edge labels.

call. In Fig. 2, the method calls to `readFile` and `writeFile` have been inlined. Any call to methods defined in `java.io.*` and `java.nio.*` is represented by a single node, and all other calls (such as the calls to `reverse` and `toString` in Line 3) get ignored.

For resolving virtual calls, we use a static algorithm that is fast but imprecise. That is, the ACG only approximates the possible sequences of calls to methods from a certain API. For example, we do not perform a proper points-to analysis while constructing the graph and indirect control-flow such as library callbacks is not tracked (but can be provided by the user). We discuss various sources of imprecision specific to our implementation later on in the evaluation.

4.2 Inter-procedural Data-Flow Graph

Our second graph is an inter-procedural data-flow graph (IDFG). Data-flow graphs are frequently used in program analysis and compiler optimization. The graph captures the flow of data between program variables without taking the control-flow into account. Each node in this graph represents a program variable or memory location. A (directed) edge between two nodes represents that data from one source node flows into the variable associated with the sink node. This way, an IDFG groups variables together that interact with each other even if they are not immediately connected in the control-flow.

The motivation of using IDFGs for finding similar programs is that programs that perform similar algorithmic tasks such as sorting or searching in collections use a similar set of base types and perform similar operations on them. In contrast to the ACG which focuses on finding programs with similar API usage, the IDFG is used to find programs that use similar algorithms. That is, while we envision a typical scenario for ACG usage as a user trying to understand how

to use a particular API, we think of the ICFG as a tool to find a method (in an API) that could be used to replace an algorithm in the user's program.

Figure 4 shows the IDFG for our example from Fig. 2. Each oval node corresponds to a program variable or a `new` expression, each octagon shaped node is a call to a procedure that could not be inlined (e.g., because it is a library call to which no source code is available). To obtain a more canonical representation, we label the nodes with the types of the program variables instead of the name. As an abstraction, we represent types that are not visible outside the current application with a question mark. We discuss possible ways of how to represent non-primitive Java types in the IDFG in the future work.

5 Evaluation

5.1 Evaluation of Code Similarity Using IDFG

We use the CodeJam dataset as a benchmark for evaluating our method of finding similar programs. This dataset consists of all the solutions to programming problems used in the Google Code Jam competition from 2008 to 2015. One advantage of using the CodeJam dataset is that the problem-solution setting acts as a free oracle for checking whether two programs are similar, since two solutions of the same problem must be input-output equivalent for the set of input test cases provided by Google (although several algorithms may exist as acceptable solutions). One downside of this dataset is that since the problems are highly algorithmic in nature (as opposed to large software design), many solution programs just operate on built-in datatypes and the only library classes that are frequently used are `String` and `StringBuffer`. This restricts our graph choices to IDFG instead of ACG. Another downside is that code quality is relatively low compared to well established open-sourced projects since the programs were produced in a competition environment with tight time constraints. An undesirable effect of this is that some programs could not be included in the experiments because of non-standard entry points (no `main` method), or other compilation issues (non-unicode characters in the source files).

Setup. We randomly selected four problems from the CodeJam dataset. There are 1280 Java programs in total in this subset. For each of the programs, we first create an inter-procedural data-flow graph as describe in Sect. 4 for its top-level entry point (usually the `main` method). Then, for each program, we use our graph kernel based method to find the top k most similar programs to it from the rest of the 1279 programs. Since these programs are known to solve one of these four problems, we consider a similar program found to be correct if it solves the same problem. Our goal of this experiment is to evaluate how well our method identifies the appropriate similar program.

Results. We first evaluate how accurately our method can find the most similar program. As a baseline comparison, a random guess would have an accuracy of 25 %. The accuracy of our method is 77.8 %. If we increase k to 2, i.e., consider the top 2 most similar programs found and check if any one of them is from the

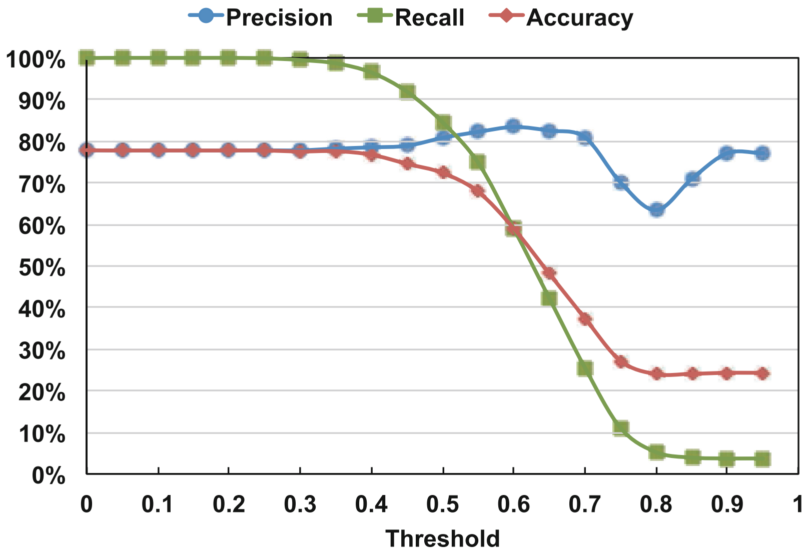


Fig. 5. Accuracy, precision and recall values plotted at different threshold values. Accuracy is the percentage times the program found is from the same problem and has a similarity score higher than the threshold, or it is from a different problem but is rejected because of a lower than threshold score. Precision is the percentage of correctly identified programs with a higher than threshold score over all programs with a higher than threshold score. Recall is the percentage of correctly identified programs with a higher than threshold score over all correctly classified programs that would have been returned without using the threshold.

same programming problem, then the accuracy increases to 87.9 %. In general, a larger k will produce better accuracy but at the expense of user experience and effort, since the user would have to spend more time reading these programs and some of them are irrelevant to her task.

We also consider using a threshold value on the similarity score for filtering the similar programs found. If the most similar program found still has a similarity score lower than the threshold, then the program is not shown to the user. Figure 5 shows the results of using different threshold values In our case, precision tops at a threshold of around 0.6. On the other hand, accuracy decreases monotonically with increasing threshold value.

Discussion. The experiments above show that our graph kernel based approach can effectively identify similar programs. However, to our surprise, a threshold value is not needed to achieve the highest accuracy (although one is needed for precision). This indicates that even a low similarity score may be sufficient to distinguish the kinds of programs, given the large space of possible implementations. We plan to include the rest of CodeJam for a more comprehensive evaluation in the future.

Threads to Validity. One internal threat to validity is the fact that several different algorithms may exist for the same programming problem, leading to widely

different implementations. While IDFG encodes semantic information of the program as data-flow, it still follows closely the structure of the program. This means IDFG will likely fail to capture the fact that different algorithms and implementations are designed to solve the same problem. An external thread to validity is that we are only using four problems in our evaluation. We are currently working on evaluating our method on the whole CodeJam dataset. However, as mentioned earlier, these programs only represent codes that are highly algorithmic in nature and they still only constitute a tiny fraction of open-sourced programs.

5.2 Evaluation of ACG Similarity

Using the ACG to find similar programs on the CodeJam corpus turned out to be infeasible. The programs in CodeJam are of a very algorithmic nature and mostly operate on the built-in types of Java. The only library classes that are frequently used are `String` and `StringBuffer` which are used to log results. Hence, building ACGs for these programs did not produce any interesting results.

To evaluate the usefulness of the ACG for finding similar programs, we set up a different experiment where we compute ACGs for every method in the Apache project `commons-lang` which is a library that provides utilities for common tasks such as handling dates or serializing objects.

We choose `commons-lang` as a benchmark because it uses large parts of the Java packages `java.lang` and `java.util` which we can use in the ACG construction. Since we do not have an oracle to decide if two methods are similar like in the case of CodeJam, the moderate size of `commons-lang` helps us to examine the similarity results by hand.

The goal of this experiment is to evaluate whether methods considered similar based on ACG isomorphisms are indeed similar. To that end, we choose the following experimental setup: for each method in `commons-lang`, we compute an ACG for all calls to methods in `java.*` excluding `String` and `StringBuffer`. We inline calls to methods inside `commons-lang` up to depth four. Method calls that exceed this limit are dropped from the ACG. While running the experiments, we experimented with inlining depth up to ten which did not change the results significantly.

If the ACG of a method has less than four nodes (i.e., two calls to library methods and source and sink), we drop it since we are only interested in methods that make at least two library calls.

Results. In total, we processed 3017 methods. Out of these, 660 methods had an ACG with at least four nodes. For each constructed ACG we identify the two most similar methods (excluding the method itself) which results in a total of 1320 pairs of ACGs together with their similarity values.

In 686 of these 1320 cases we found a real isomorphism (with a similarity of 100 %). In 4 cases, we found a similarity between 99 and 80 %, in 8 cases a similarity between 79 and 60 %, in 24 cases between 59 and 40 %, and all other similarities were below 40 %.

The most notable part of our experimental result is the high number of real isomorphisms (over 50 %). To get an intuition where these isomorphisms come from, we investigated 40 methods by hand. In 27 of the 40 cases, the graphs were isomorphic because both methods only contained a single statement calling the same method. Hence, due to inlining, the ACGs were identical. In the remaining cases, the methods indeed used the same set of library calls such as iterators over collections or modifications of `Date` objects. We emphasize that this high number of isomorphisms is a result of our experimental setup and this would not be the case when searching for similar methods in a different code base.

In the four cases where methods had a similarity between 99 and 80 %, the methods were slightly different but shared a number of API calls. One example of this is the similarity between `DateUtils.round` and `DateUtils.ceilling`. The `round` method calls to another method that is almost identical to `ceiling` (this method and `ceiling` are in fact isomorphic).

In the similarity range between 79 and 60 %, we still find interesting results. Examples of similar procedures are pairs like `FormatCache.getDateTime-Instance` and `FormatCache.getTimeInstance` which, without going into the details of the code are understandably similar by looking at their names. Five of the eight similarities between 79 and 60 % were cases where one method called to other.

Similarities between 59 and 40 % were, for example, found in the `StringUtils` class between methods like `endsWithAny` and `startsWithAny`, or `removeEndIgnoreCase` and `removeStartIgnoreCase` which, as their naming suggests, perform very similar tasks. Other cases of 50 % similarity are different methods to find threads in `ThreadUtil`. These methods all iterate over a collection of `Thread` objects but use different methods to filter this set.

Even methods with a similarity below 40 % still were interesting in many cases. For example for the method `LocalUtils.countriesByLanguage` the most similar method was `LocalUtils.languagesByCountry`, or for `Fraction.divideBy` the most similar method (with 18 % similarity) was `Fraction.multiply`. Only when we reach a similarity of below 15 %, the results become less useful.

Discussion. This experiment shows that similarity based on the ACG is indeed useful to identify methods that serve a similar purpose. We emphasize that our experimental setup of finding similarities in the same code-base is certainly biased towards finding many isomorphic graphs, so the success rate of finding similar code with this approach can not be generalized from this experiment. What the experiment shows, however, is that methods with a similarity between 99 % and 20 % are still very similar even if they do not share code and that the approach hardly produces false alarms. For the method pairs that we inspected, there was no case where we could not spot the similarity.

Threats to Validity. Several threats to validity have to be discussed in this experiment. We already mentioned that searching for similar methods in the same code base is biased towards finding many isomorphic graphs. For a less biased experimental setup, we would need labeled data like in the case of CodeJam. We would

need an oracle that can decide if methods are similar to measure how often our approach does not find a similar method where one exists. Unfortunately, we do not have such an oracle but it is part of our future work to build up a corpus to further evaluate our approach. Another thread to validity is the choice of `java.*` as an API and `commons-lang` as code base. In the future work we will experiment with more code bases and different APIs but within the scope of this paper we believe that this experiment is sufficient to convey our idea.

6 Related Work

The problem of finding similarities in source code is a known problem in software engineering. It crops up in many software engineering contexts as diverse as program compression [5], malware similarity analysis [3], software theft detection [14], software maintenance [7], and Internet-scale code clone search [25].

Previous research in program similarity has focused more on detecting syntactic similarity [24] and less on detecting semantic similarity, as the latter is generally undecidable. We can classify these different approaches into five categories: Text-based, Token-based, Tree-based, Semantic-based, and Hybrid.

In text-based solutions, the source code of a program is divided into strings and then compared against another. Under this type of solution, two programs are similar if their strings match [2]. In token-based solutions (lexical), the source code of a program is transformed into a sequence of lexical tokens using compiler style lexical analysis. The produced sequences are then scanned for duplicated subsequences of tokens. The representative work here is Baker's token based clone detection [2]. In tree-based solutions, the source code of a program is parsed in order to produce an abstract syntax tree (AST). The produced AST is then scanned for similar subtrees. The representative work here is Jiang's *Deckard* [10]. In semantic-based solutions, a source code is statically analyzed to produce a program dependency graph (PDG) [6]. Then, the program similarity problem is reduced to the problem of finding isomorphic graphs using program slicing [13]. In hybrid solutions, both syntactic and semantic characteristics are used to find similar code. The representative work here is Leitao's hybrid approach for detecting similar code. This hybrid approach combines syntactic techniques based on AST metrics, semantic techniques (call graphs), and specialized comparison functions to uncover code redundancies [15].

The approach presented in this paper can be seen as a hybrid solution as well. It identifies similar programs using graph similarity like semantic-based solutions but is agnostic to the kind of graph that is being used. For example our ACG is more of a syntactic representation and the IDFG more a semantic representation of the program.

When considering similarity measures of graphs one has to carefully distinguish between measures that are applied to labeled graphs and measure applied to unlabeled graphs. A labeled measure d may use information of the vertex names. For example the edit distance is usually defined for two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ over the same vertex set as $d(G_1, G_2) :=$

$|E_1 \setminus E_2| + |E_2 \setminus E_1|$. It captures the amount of edges/non-edges that need to be altered to turn the one graph into the other. In contrast to this, a measure for unlabeled graphs is not allowed to depend on the names of the vertices, that is for a permutation π of the vertices of G_2 we must have that $d(G_1, G_2) = d(G_1, \pi(G_2))$.

An overview over some graphs similarity measures is given in [26]. Labeled graph similarity measures are not suitable for our intended applications. Indeed, the names of the vertices of the IDFG and ACG do not seem to carry relevant information. (This is different for the labels that are assigned by the algorithm, which carry structural information, as discussed earlier.) The question remains which unlabeled graph similarity measures are suitable to capture code similarity? From a conceptual point it appears that the occurrence of substructures of certain kinds in a node's vicinity is related to the purpose of a code snippet containing said node. This is supported by the findings in [16,20–22]. Guided by this insight, we chose the similarity based on Weisfeiler-Leman algorithm for our purposes. While it detects similar information as subgraph counts, it provides us with two significant advantages. On the one hand it is very efficiently computable, which not the case for subgraph detection, as also explained in [16,22]. On the other hand it easily allows us to exploit the label information of different types of graphs generated from programs.

7 Conclusion

We have presented a generic algorithm to compute program similarity based on the Weisfeiler-Leman graph kernels. Our experiments suggest that the algorithm performs well for the IDFG and ACG representation of Java code that we proposed. However, we believe that our algorithm will also perform well with other graph-based models.

We see several interesting applications that we want to pursue as future work: one interesting property of graph kernels is that a combination of two graph kernels is again a graph kernel. In the case of the ACG, this would allow us to compute separate ACGs for different APIs (e.g., `java.lang` and `java.util`) and either use them in isolation or combine them, which would allow us to build more efficient search algorithms. Another interesting application would be to combine kernels from entirely different graphs, such as the ACG and the IDFG to experiment with new concepts of similarity.

Further, using graph kernels makes it easy to experiment with different graph representations. One could for example use a simplified version of a control-flow graph or use a more abstract labeling of the nodes to model different kinds of program similarity.

References

1. Babai, L., Erdős, P., Selkow, S.M.: Random graph isomorphism. SIAM J. Comput. **9**(3), 628–635 (1980)
2. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: 2nd Working Conference on Reverse Engineering, WCRE 1995, Toronto, Canada, 14–16 July 2005, pp. 86–95 (1995)
3. Cesare, S., Xiang, Y.: Software Similarity and Classification. Springer Briefs in Computer Science. Springer, London (2012)
4. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for CNF. In: Malik, S., Fix, L., Kahng, A.B. (eds.), Proceedings of the 41th Design Automation Conference, DAC, San Diego, CA, USA, 7–11 June 2004, pp. 530–534. ACM (2004)
5. Evans, W.S.: Program compression. In: Koschke, R., Merlo, E., Walenstein, A. (eds.) Duplication, Redundancy, and Similarity in Software, 23–26 July 2006, vol. 06301 of Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
6. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)
7. Godfrey, M.W., Zou, L.: Using origin analysis to detect merging and splitting of source code entities. IEEE Trans. Softw. Eng. **31**(2), 166–181 (2005)
8. Grohe, M.: Fixed-point definability and polynomial time on graphs with excluded minors. J. ACM **59**(5), 27 (2012)
9. Horváth, T., Gärtner, T., Wrobel, S.: Cyclic pattern kernels for predictive graph mining. In: Kim, W., Kohavi, R., Gehrke, J., DuMouchel, W. (eds.) Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, 22–25 August 2004, pp. 158–167. ACM (2004)
10. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 96–105. IEEE Computer Society Washington, DC, USA (2007)
11. Junttila, T.A., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX, New Orleans, Louisiana, USA, 6 January 2007. SIAM (2007)
12. Ke, Y., Stolee, K.T., Le Goues, C., Brun, Y.: Repairing programs with semantic code search. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 295–306, Lincoln, NE, USA, November 2015. doi:10.1109/ASE.2015.60, http://people.cs.umass.edu/brun/pubs/pubs/Ke15ase.pdf
13. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 40–56. Springer, Heidelberg (2001)
14. Lancaster, T., Culwin, F.: A comparison of source code plagiarism detection engines. Comput. Sci. Edu. **14**(2), 101–112 (2004)
15. Leitão, A.M.: Detection of redundant code using R2D2. In: 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003), Amsterdam, The Netherlands, 26–27 September 2003, pp. 183–192 (2003)
16. Lestringant, P., Guihéry, F., Fouque, P.-A.: Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS 2015, pp. 203–214. ACM, New York (2015)

17. Mahmoud, A., Bradshaw, G.: Estimating semantic relatedness in source code. ACM Trans. Softw. Eng. Methodol. **25**(1), 10:1–10:35 (2015)
18. McKay, B.D., Piperno, A.: Nauty and traces user guide. https://cs.anu.edu.au/people/Brendan.McKay/nauty/nug25.pdf
19. Pikhurko, O., Verbitsky, O.: Logical complexity of graphs: a survey. CoRR, abs/1003.4865 (2010)
20. Pradhan, P., Dwivedi, A.K., Rath, S.K.: Detection of design pattern using graph isomorphism and normalized cross correlation. In: Parashar, M., Ramesh, T., Zola, J., Narendra, N.C., Kothapalli, K., Amudha, J., Bangalore, P., Gupta, D., Pathak, A., Chaudhary, S., Dinesha, K.V., Prasad, S.K. (eds.) Eighth International Conference on Contemporary Computing, IC3, Noida, India, 20–22 August 2015, pp. 208–213. IEEE Computer Society (2015)
21. Qiu, J., Su, X., Ma, P.: Library functions identification in binary code by using graph isomorphism testings. In: Guéhéneuc, Y., Adams, B., Serebrenik, A. (eds.) 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER, Montreal, QC, Canada, 2–6 March 2015, pp. 261–270. IEEE (2015)
22. Qiu, J., Su, X., Ma, P.: Using reduced execution flow graph to identify library functions in binary code. IEEE Trans. Softw. Eng. **42**(2), 187–202 (2015)
23. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from "big code". In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, pp. 111–124. ACM, New York (2015)
24. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Sci. Comput. Program. **74**(7), 470–495 (2009)
25. Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., Lopes, C.V.: SourcererCC: scaling code clone detection to big code. CoRR, abs/1512.06448 (2015)
26. Schweitzer, P.: Isomorphism of (mis)labeled graphs. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 370–381. Springer, Heidelberg (2011)
27. Shervashidze, N., Schweitzer, P., van Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler-lehman graph kernels. J. Mach. Learn. Res. **12**, 2539–2561 (2011)
28. Shervashidze, N., Vishwanathan, S.V.N., Petri, T., Mehlhorn, K., Borgwardt, K.M.: Efficient graphlet kernels for large graph comparison. In: Dyk, D.A.V., Welling, M. (eds.) Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, AISTATS, Clearwater Beach, Florida, USA, 16–18 April 2009, vol. 5 of JMLR Proceedings, pp. 488–495. JMLR.org (2009)
29. Sidiroglou-Douskos, S., Lahtinen, E., Long, F., Rinard, M.: Automatic error elimination by horizontal code transfer across multiple applications. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, pp. 43–54. ACM, New York (2015)
30. Stolee, K.T., Elbaum, S., Dobos, D.: Solving the search for source code. ACM Trans. Softw. Eng. Methodol. **23**(3), 26:1–26:45 (2014)
31. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1999, p. 13. IBM Press (1999)