An Experimental Analysis of Graph-Distance Algorithms for Comparing API Usages

Sebastian Nielebock
Otto-von-Guericke University
Magdeburg, Germany
sebastian.nielebock@ovgu.de

Paul Blockhaus
Otto-von-Guericke University
Magdeburg, Germany
paul.blockhaus@ovgu.de

Jacob Krüger
Ruhr-University Bochum,
Germany
jacob.krueger@rub.de

Frank Ortmeier
Otto-von-Guericke University
Magdeburg, Germany
frank.ortmeier@ovgu.de

***Abstract*—Modern software development heavily relies on the reuse of functionalities through Application Programming Interfaces (APIs). However, client developers can have issues identifying the correct usage of a certain API, causing misuses accompanied by software crashes or usability bugs. Therefore, researchers have aimed at identifying API misuses automatically by comparing client code usages to correct API usages. Some techniques rely on certain API-specific graph-based data structures to improve the abstract representation of API usages. Such techniques need to compare graphs, for instance, by computing distance metrics based on the minimal graph edit distance or the largest common subgraphs, whose computations are known to be NP-hard problems. Fortunately, there exist many abstractions for simplifying graph distance computation. However, their applicability for comparing graph representations of API usages has not been analyzed. In this paper, we provide a comparison of different distance algorithms of API-usage graphs regarding correctness and runtime. Particularly, correctness relates to the algorithms' ability to identify similar correct API usages, but also to discriminate similar correct and false usages as well as non-similar usages. For this purpose, we systematically identified a set of eight graph-based distance algorithms and applied them on two datasets of real-world API usages and misuses. Interestingly, our results suggest that existing distance algorithms are not reliable for comparing API usage graphs. To improve on this situation, we identified and discuss the algorithms' issues, based on which we formulate hypotheses to initiate research on overcoming them.**

***Index Terms*—API usage, graph similarity, misuse**

I. Introduction

Modern software development heavily relies on reusing existing software to effectively and efficiently construct desired products. Software reuse can include copying & pasting code from other locations or discussion forums (e.g., StackOverflow), (internal) software platforms or product lines, and the integration of specified Application Programming Interfaces (APIs) of (external) software libraries in an ecosystem [11], [31], [32]. Especially the latter is vulnerable to be misused by client developers. For instance, a developer may use a method of an API differently than expected by the API developers (e.g., parameters contradicting an implicit specification of that method). We call cases in which this causes unexpected negative behavior of the software *API misuses*, which may manifest as crashes, usability problems, or security issues [5], [40], [41], [49]. In this paper, we consider API usages as correct or incorrect usages (i.e., misuses), and their comparison as a way to discriminate similar correct API usages from similar misuses as well as to distinguish completely different API usages.

We focus on API misuses since they are a prevalent issue in software development. For example, studies show that approximately half of the bug fixes in five open-source projects require an adaptation of API usages [67] and more than half of 806 projects use outdated APIs [64] which may cause security issues. To describe such API misuses, Amann et al. [4] have defined a corresponding taxonomy. Other studies identified root causes of API misuses, for instance, the absence of proper API documentation, APIs that were too complex, a lack of domain knowledge, backward incompatibilities, issues with the execution environment, or a lack of communication channels between API and client developers [21], [25], [33], [41], [42], [50], [53], [54], [66].

Researchers focus on two directions to deal with API misuses: First, avoiding them by mitigating the above causes, for instance, using automated tools to enhance the documentation [63]. Second, automatically detecting API misuses. In this context, techniques for mining and comparing specifications of API usages against the suspicious API client code are prevalent. These specifications may be represented as formal specification, such as finite-state automata [8], [19], [65] and dynamic invariants [18], or as patterns of API usages [3], [5], [36], [45], [47], [62].

We focus on the second direction, which typically requires comparisons of different API usages. This encompasses searching and mining similar API usages [15], [38], [44], [48], [58], comparing pattern candidates [5], [45], or detecting pattern violations [5], [26], [48]. Since API usages are more and more represented as graphs [5], [26], [45], [47], this essentially means to compare the distance of graphs. However, established graph-distance algorithms, such as computing the minimal graph edit distance (GED) or the maximum common subgraph, are NP-hard. Fortunately, advances have been made to relax or approximate distance computation. Still, to the best of our knowledge, the applicability of these algorithms for comparing API usages has not been systematically analyzed.

In this paper, we address this gap by analyzing a set of well-known graph-distance algorithms and investigating whether they are feasible to compare API usage graphs. We consider a "good" distance algorithm to be *effective* and *efficient*. For detecting API misuses, an effective algorithm is able to compute a distance that significantly differs when comparing two correct (or two incorrect) usages rather than comparing a correct usage and a misuse. So, we can discriminate correct from incorrect

```
1  package pkg.at.some.loc;
2
3  import from.another.place.Foo;
4  import from.another.place.Bar;
5  import from.another.place.Baz;
6
7  public class AUGSample {
8      Foo fooObj = new Foo(42,"text");
9
10     public Integer computeSomething(Bar barObj) {
11         Baz bazObj = new Baz(this.fooObj);
12         if(bazObj.hasCharacteristic()){
13             bazObj.doSomething(barObj);
14         }
15         return bazObj.getResult();;
16     }
17 }
```

Listing 1. Code example for the AUG in Figure 1.

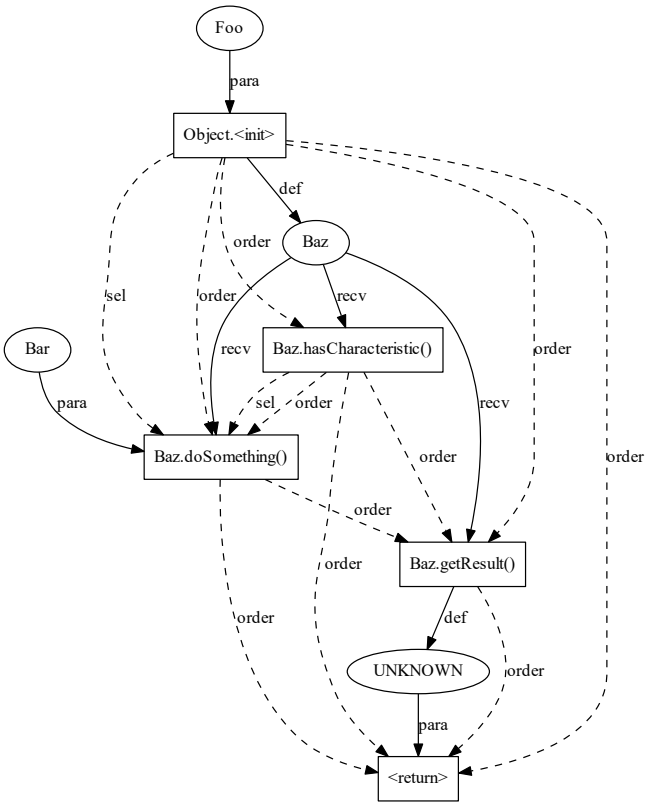

Fig. 1. AUG representing the method `computeSomething` in Listing 1.

(i.e., misused) API usage, and effectively reduce the false positive rate of misuse detectors, a well-known issue in static API misuse detection [7], [35]. For searching similar API usages, the algorithm should compute a low distance value for API usages of the same API in similar contexts. This way, the algorithm can help to effectively filter API usages for subsequent pattern mining. Since pattern mining usually requires multiple thousands to millions of comparisons, the algorithm must efficiently compute the distance of real-world graphs. While we do not expect that the comparison is interactively usable (i.e., done in a fraction of seconds), it should be efficient enough to compare several thousands of graphs in a matter of minutes. This is comparable to automated tests executed in a continuous integration system [68].

For our experimental comparison, we systematically identified eight graph-distance algorithms and selected those able to achieve our goals. Then, we selected two different datasets of API misuses and correct usages and transformed each entry into an *API Usage Graph* (AUG), an established data structure proposed by Amann et al. [5], [7] (cf. Section II), as well as into so-called API misuse correction rules (cf. Section II-B) that we have proposed [47]. We computed the distances between all AUGs (i.e., correct to correct usages, misused to misused, misused to correct usages, and vice versa) within a framework we implemented. Based on the distribution of the resulting distance values, we determined the ability of the selected algorithms to effectively discriminate misused from correct API usages. We discuss our main results as well as their implications in Section IV, and publish our data, results, and experimental framework in our replication package.[1]

II. API Usage Representations

In this section, we describe the basic concepts required to understand our experimental comparison.

A. API Usage Graphs (AUG)

Source code can be represented by different data structures, such as text, token streams, Abstract Syntax Trees (ASTs), control flow graphs, or Program Dependency Graphs (PDGs). For API misuse detection, Amann et al. [5], [7] developed the AUG. It is tailored to represent the specificities of API usages, in contrast to, for instance, general-purpose PDGs. Using AUGs as pattern representations of API usages, Amann et al. achieved a higher precision and recall than comparable algorithms for detecting API misuses. In their recent work, Kang et al. [26] defined *extended AUGs (eAUGs)* to further include specific properties of API usages as additional nodes and edges. They found that using these extensions together with active learning had positive effects on misuse detection. However, for our experimental comparison of graph-distance algorithms, the simpler original AUGs are sufficient since these algorithms do not discriminate between different node and edge types. Note that we target the Java programming language with its specific elements, and thus we limit our descriptions to Java, too. As a running example, we use an AUG (cf. Figure 1) based on the method `computeSomething` starting in line 10 of Listing 1.

In general, an AUG represents a directed, labeled multigraph $aug := (V, E, \Sigma_V, \Sigma_E, s, t, l_V, l_E)$ where V is a set of vertices or nodes, $E : V \times V$ is a multiset of edges, Σ_V and Σ_E are finite alphabets of node (V) and edge (E) labels, $s : E \to V$ and $t : E \to V$ are functions to map an edge to its source (s) or target (t) node, and $l_V : V \to \Sigma_V$ and $l_E : E \to \Sigma_E$ are the node and edge labeling functions, respectively. As we can see in Figure 1, AUGs may involve multiple types of nodes and edges. Nodes can represent actions (i.e., rectangles in the graphical representation) or data (i.e., ellipses in the graphical representation). An action node describes an API

[1] https://doi.org/10.5281/zenodo.5255402

method call (e.g., node `Baz.doSomething()`) or a control structure (e.g., node `<return>`). A data node represents a raw value or an object instance (e.g., node `Baz`). There exist several subtypes of these nodes. To determine these node types, we define the function $type : V \rightarrow String$, which returns the type of a particular node as a String value.

An edge can represent either the data-flow (solid arrow) or the control-flow (dashed arrow). Data-flow edges can show the usage of a data node as a parameter (e.g., the `para` edge directing from node `Bar` to the node `Baz.doSomething()`), an object instance on which a method is called (e.g., the `recv` edge directing from node `Baz` to `Baz.doSomething()`), or the creation of a new object instance (e.g., the `def` edge directing from node `Object.<init>` to node `Baz`). Control-flow edges can show a condition (e.g., the `sel` edge directing from node `Baz.hasCharacteristic` to node `Baz.doSomething`) or certain execution orders (e.g., the `order` edge between node `Baz.doSomething` and `Baz.getResult`). For further details on node and edge types, we refer to Sven Amann's dissertation [5].

Since AUGs are generated based on static ASTs, control-flow information and type resolution are limited. Particularly, `order`-edges are generated conservatively, namely as the transitive closure of all `order`-edges between each pair of action nodes. To enable type resolution, the AUG generation requires access to the source code or library (i.e., of the used API) to find the declarative type of, for instance, a certain method. If this is not provided, the AUG generation uses the type `UNKNOWN`. Note that the `UNKNOWN`-node in Figure 1 cannot be resolved, since the generation did not have access to the declaration of the method `Baz.getResult()`. Thus, it cannot decide whether the object (i.e., the return type of the method `computeSomething()`) is of type `Integer` or of a subtype of `Integer`. Also, the generation fails to correctly resolve dynamically inferred types, such as generic types in Java.

Some graph-distance algorithms rely on node and edge labels to compute similarity, which is why the labeling functions l_E (i.e., edge labeling) and l_V (i.e., node labeling) are important [5]. Regarding edges, l_E assigns edges their respective types, and thus $|\Sigma_E|$ denotes the number of different edge type names. Labels of action nodes describe API method calls, such as `Baz.doSomething()`. This consists of the declaring type name of the called method (i.e., `Baz`) and the method name (i.e., `doSomething()`). Note that parameters are not part of the label, since they are represented by additional nodes connected with `para`-edges. In case of specialized action nodes, for instance, return statements (i.e., `<return>`) or object constructors (i.e., `<init>`), special labels are defined. For an overview of these label types, we refer to the original work of Sven Amann [5]. Regarding data nodes, the resolved declaring type names are used as labels.

For our analysis, we adapted the standard definition twice. First, we used slightly different labels for data nodes. Namely, we label nodes representing raw values of primitive types (e.g., `int`, `String`) with the actual value. Our rationale is that these values may indicate a certain meaning, which would be hidden when abstracting them with the declaring type name. For example, the method `getInstance` from the class `java.security.MessageDigest`[2] requires a `String` representing the hashing algorithm as input. Second, we define a function $api : V \rightarrow String$ that returns the complete declaring type name if it is resolvable (e.g. `java.lang.Object` for the node `Object.<init>`). If the type cannot be resolved, the String represents the type name or an empty String in case no type is apparent.

B. AUG Correction Rules

In our previous work, we introduced the notion of *correction rules* [47]: an AUG-based encoding of changes needed to fix an API misuse, which can be automatically generated from fixing commits. The goal was to transfer the knowledge of one API-misuse fix to similar usages in other projects. We show an example of such a rule in Figure 2. This rule describes a case in which a developer forgot to call the condition check in line 12 in Listing 1 (i.e., `if(bazObj.hasCharacteristic())`), and defines how to add this call to fix this misuse. Each rule consists of two AUGs: a misuse AUG (left) and a fix AUG (right). Furthermore, correction rules describe the changes needed to transform the misuse into the fix. For that purpose, we computed the minimal mapping (depicted as blue `transform`-edges) between the two AUGs using the Kuhn-Munkres algorithm [39]. To simplify the rule, we left out nodes (with their respective edges) that are not affected by any change. For instance, the nodes `Foo` and `Bar` in Figure 1 are not part of the rule in Figure 2. Note that for real examples, depending on the committed changes, the number of left out nodes is usually much higher. An addition, like the added conditional statement, is represented by mapping a so-called empty node (i.e., ϵ) in the misuse AUG to the respective added node (e.g., `Baz.hasCharacteristics()`). A deletion is represented vice-versa. From such correction rules, we can derive related misuse and fix AUGs that we can compare to other usages, allowing us to evaluate graph-distance algorithms.

III. Methodology

In the following, we describe the methodology we employed for our experimental analysis.

A. Goal of the Distance Computation

We define $dist$ to be a distance function taking as input two AUGs aug_i and aug_j to compute a normalized distance, namely: $dist(aug_i, aug_j) \in [0, 1]$. 0 denotes that the two usages described by the AUGs are identical, while 1 means that the usages are most dissimilar (i.e., completely different API calls). We distinguish between two types of API usages, and thus of AUGs: correct usages (i.e., aug_c) and misuses (i.e., aug_m). Moreover, we define two reference usages described by their respective AUGs: a correct one aug_{rc} and a misuse aug_{rm}. The goal of $dist$ is to compute a distance between a reference AUG and another AUG so that distances between

[2]https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/security/MessageDigest.html

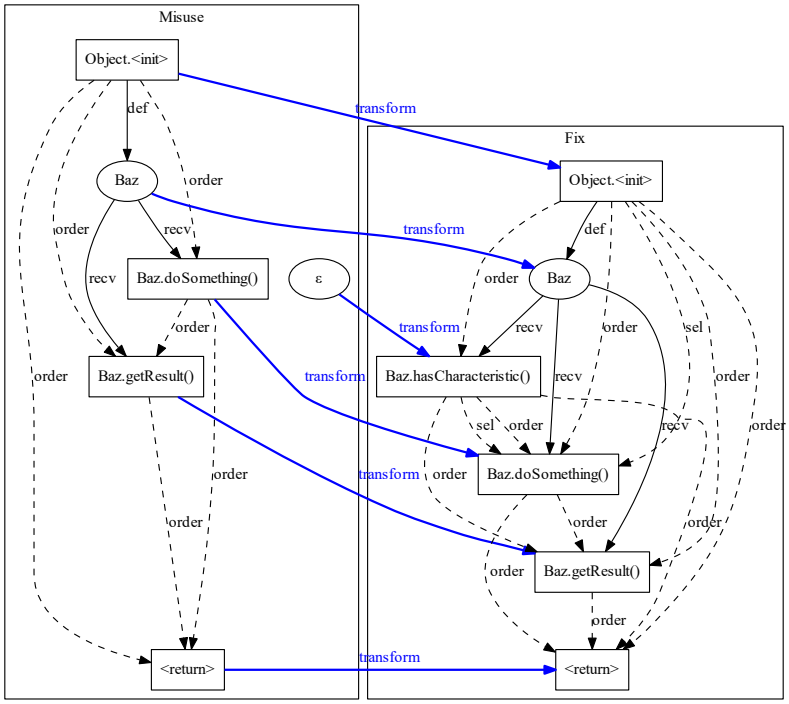

Fig. 2. AUG correction rule by adding the condition in line 12 in Listing 1.

similar types (i.e., correct usages of the same API) have a smaller distance than two dissimilar types (i.e., a correct usage and a misuse). Thus, we expect:

$$\begin{aligned} dist(aug_{rc}, aug_c) &< dist(aug_{rc}, aug_m) \\ dist(aug_{rm}, aug_c) &> dist(aug_{rm}, aug_m) \end{aligned} \tag{1}$$

Furthermore, we assume that the reference usages are based on a correction rule (cf. Section II-B). We denote this rule as $aug_{rm} \rightarrow aug_{rc}$, where aug_{rm} is the misuse and aug_{rc} is its respective fix. To discriminate misuses from correct usages, in addition to Equation 1, $dist$ should satisfy:

$$\begin{aligned} dist(aug_{rc}, aug_c) &< dist(aug_{rm}, aug_c) \\ dist(aug_{rc}, aug_m) &> dist(aug_{rm}, aug_m) \end{aligned} \tag{2}$$

So, we can use a correction rule to compute the distance from both, the misuse and the correct usage, to any arbitrary usage. If the distance is lower for the misuse than for the correct part, the usage is marked as misuse. Ideally, the correction rule could then be used as a patch to fix the misuse.

To assess such rules, we assume a dataset of AUGs describing correct usages $C = \{aug_{c1}, aug_{c2}, \cdots, aug_{cm}\}$ and a set of misuses $M = \{aug_{m1}, aug_{m2}, \cdots, aug_{mn}\}$. We denote a rule to be *applicable* with a distance function $dist$, if it satisfies Equations 3, 4, 5, and 6:

$$\frac{\sum_{aug_c \in C} dist(aug_{rc}, aug_c)}{|C|} < \frac{\sum_{aug_m \in M} dist(aug_{rc}, aug_m)}{|M|} \tag{3}$$

$$\frac{\sum_{aug_c \in C} dist(aug_{rm}, aug_c)}{|C|} > \frac{\sum_{aug_m \in M} dist(aug_{rm}, aug_m)}{|M|} \tag{4}$$

$$\frac{\sum_{aug_c \in C} dist(aug_{rc}, aug_c)}{|C|} < \frac{\sum_{aug_c \in C} dist(aug_{rm}, aug_c)}{|C|} \tag{5}$$

$$\frac{\sum_{aug_m \in M} dist(aug_{rc}, aug_m)}{|M|} > \frac{\sum_{aug_m \in M} dist(aug_{rm}, aug_m)}{|M|} \tag{6}$$

These equations generalize Equations 1 and 2 by comparing the average distance over a set of other usages. More generally, an *applicable* rule produces significantly different distributions of distance values for one set (e.g., $dist(aug_{rc}, aug_c)$) than for another set (e.g., $dist(aug_{rc}, aug_m)$).

Our goal is to find a distance function $dist$ that maximizes the number of *applicable* rules on a set of generated correction rules based on a given dataset of correct and misuse AUGs. However, this problem is not trivial, since existing graph-distance metrics usually correlate with general graph similarity. This can be distorted if, for example, two AUGs share only a small fraction of a similar usage (e.g., a usage in a different context), and thus still lead to a too large distance value. Then, the differences between a misuse and a correct AUG cannot be determined. Another issue is that two usages may represent two completely different cases, but the equations may still be randomly satisfied. Finally, two correct usages may also represent two alternative solutions for the same misuse, so that the distance falsely indicates a misuse (i.e., false positive).

B. Graphs-Distance Algorithms

For our comparison, we considered different graph-distance algorithms that we identified by reviewing surveys on graph similarity [13], code clone detection [30], [55], and binary code similarity [61]. In addition, we checked a curated list of binary graph-distance algorithms available on GitHub.[3] We focused on

[3]https://github.com/SystemSecurityStorm/Awesome-Binary-Similarity

algorithms that compute a distance between graphs. Particularly, we expect the underlying distance metrics to be applicable on AUGs, namely directed labeled multigraphs as defined in Section II-A. Moreover, we expect the distance metrics to describe a relative distance (i.e., $dist(aug_1, aug_2) \in [0, 1]$) to easily compare different algorithms. Furthermore, we selected those algorithms that directly and only compute the distance based on the two compared AUGs. This way, we aimed to avoid that knowledge from other AUG comparisons is required. For instance, a machine-learning algorithm may learn certain features from previous comparisons, helping it to better discriminate other AUGs. However, for such an algorithm, we can hardly decide (also compared to other algorithms) whether the results are based on the algorithm itself or the training data. Finally, we required an existing implementation that we could apply in our experiment (e.g., via an API) or a sufficiently detailed description of the algorithm to easily re-implement it. Based on these criteria, we selected the four underlying metrics we introduce in the following. Note that we used two different versions for the GED and four different versions of the Exas vector algorithms in our experiments (cf. Section IV). For the sake of simplicity, we describe these metrics based on the distance of the two AUGs $aug_A := (V_A, E_A, \Sigma_{V_A}, \Sigma_{E_A}, s_A, t_A, l_{V_A}, l_{E_A})$ and $aug_B := (V_B, E_B, \Sigma_{V_B}, \Sigma_{E_B}, s_B, t_B, l_{V_B}, l_{E_B})$.

Graph Edit Distance (GED). The Graph Edit Distance (GED) describes the minimal costs of edit operations (i.e., replacements, insertions, and deletions of nodes and edges) to transform one graph into another [57]. It is widely used to compute inexact matchings of structurally similar graphs, and thus applies for our use case [61]. A critical factor to represent a meaningful GED is a properly chosen cost function to determine the mapping between two graphs [59]. Assume $i, j \in V_A$ and $k, l \in V_B$ to be nodes of the two AUGs, and $ij \in E_A$ and $kl \in E_B$ to represent their edges. To compute the GED on AUGs, we selected the following cost functions for node replacement (Equation 7), node deletion and addition (Equation 8), edge replacement (Equation 9), as well as edge deletion and addition (Equation 10).

$$cost_r(i,k) = \begin{cases} 0 & \text{if } l_{V_A}(i) = l_{V_B}(k) \wedge type(i) = type(k) \\ 1 & \text{if } type(i) = type(k) \\ 2 & \text{otherwise} \end{cases} \tag{7}$$

$$cost_d(i) = cost_a(k) = 2 \tag{8}$$

$$cost_r(ij,kl) = \begin{cases} 0 & \text{if } l_{E_A}(ij) = l_{E_B}(kl) \\ 2 & \text{otherwise} \end{cases} \tag{9}$$

$$cost_d(ij) = cost_a(kl) = 2 \tag{10}$$

Regarding the edge costs (i.e., Equation 9), we do not need to separate costs for individual types, since the label function l_E already denotes these types (cf. Section II-A).

Due to the variety of different node and edge types in AUGs, one may define a large number of different cost functions to account for their specific properties. For the sake of simplicity, we performed only small modifications to the cost function (i.e., equal costs for all edit operations) and handle all nodes identically. Note that our cost functions still satisfy the triangle inequality: $cost_d(i) + cost_a(j) \geq cost_r(i,j)$ and $cost_d(ij) + cost_a(kl) \geq cost_r(ij,kl)$. The GED is then defined over all possible sequences of edit operations transforming aug_A into aug_B, with the function ged returning the minimal edit costs. To normalize the distance, we define the maximum costs between nodes (i.e., Equation 11) and edges (i.e., Equation 12).

$$mcost_n = \max_{\forall i \in V_A, k \in V_B} \{cost_r(i,k), cost_d(i,k), cost_a(i,k)\} \tag{11}$$

$$mcost_e = \max_{\forall ij \in E_A, kl \in E_B} \{cost_r(ij,kl), cost_d(ij,kl), cost_a(ij,kl)\} \tag{12}$$

For our cost definition, this means $mcost_n = mcost_e = 2$. The normalized distance is then defined as:

$$dist_{ged}(aug_A, aug_B) = \frac{ged(aug_A, aug_B)}{max(|V_A|, |V_B|) \cdot mcost_n + max(|E_A|, |E_B|) \cdot mcost_e} \tag{13}$$

The exact computation of $dist_{ged}$ is known to be NP-hard, and thus only applicable for small graphs. For this reason, we applied two algorithms using heuristics to compute an almost exact GED. First, the algorithm of Abu-Aisheh et al. [1] uses a simplified version of the well-known A*-algorithm, building on a depth-first search together with a pruning technique to discard edit sequences with high costs. Their algorithm is implemented in the NetworkX python library,[4] which is why we refer to it as `NetworkXGED`.

Second, we applied the Hungarian algorithm (also known as Kuhn-Munkres algorithm) [39]. This algorithm computes a one-to-one mapping between the nodes of each partition in a bipartite graph, producing minimal edit costs for those nodes. Then, this mapping is used to compute the GED. Since this algorithm only considers the costs for node edits, we set $mcost_e = 0$. To compute the minimal mapping, we applied the linear sum assignment implemented in the python library scipy,[5] which is based on the description of Crouse [16]. We refer to this algorithm as `HungarianGED`.

Maximum Common Subgraph (MCS). Another distance metric is the maximum common subgraph [14] (MCS). It is based on the notion that similar graphs share a larger common subgraph. A major advantage of this metric is that, in contrast to the GED, it does not require a carefully designed cost function to get a valid and meaningful distance. Thus, the MCS qualifies as another candidate to measure the structural similarity of AUGs. Still, it has been shown that the MCS can be calculated with any GED algorithm using the following cost functions for node replacement (Equation 14), node deletion and addition (Equation 15), edge replacement (Equation 16), as well as edge deletion and addition (Equation 17) [12]:

$$cost_r(i,k) = \begin{cases} 0 & \text{if } l_{V_A}(i) = l_{V_B}(k) \wedge type(i) = type(k) \\ \infty & \text{otherwise} \end{cases} \tag{14}$$

[4]https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.similarity.graph_edit_distance.html

[5]https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear_sum_assignment.html

$$cost_d(i) = cost_a(k) = 1 \tag{15}$$

$$cost_r(ij, kl) = \begin{cases} 0 & \text{if } l_{E_A}(ij) = l_{E_B}(kl) \\ \infty & \text{otherwise} \end{cases} \tag{16}$$

$$cost_d(ij) = cost_a(kl) = 1 \tag{17}$$

Similar to the ged function, the mcs function computes the minimal costs to produce the maximum common subgraph. To obtain a normalized distance between $[0, 1]$, our $dist_{mcs}$ function is defined as follows:

$$dist_{mcs}(aug_A, aug_B) = \frac{mcs(aug_A, aug_B)}{max(|V_A|, |V_B|) + max(|E_A|, |E_B|)} \tag{18}$$

Since computing the MCS is also NP-hard, we reuse `HungarianGED` as `HungarianMCS` to compute the maximum common subgraph. This algorithm includes only node-related costs, and thus we set the denominator of Equation 18 to $max(|V_A|, |V_B|)$ (i.e., ignoring edge costs).

Node-Node Similarity. Another metric for graph similarity is the link-based similarity of nodes originating from hyperlinked environments, such as the world wide web [28]. However, most algorithms compute similar nodes within one graph only. To compare between graphs, Blondel et al. [10] proposed a vertex similarity. Particularly, they compute a node-node similarity matrix S as the limit of a normalized iterative matrix multiplication with an even number of iterations. In each iteration, the following formula is applied:

$$S_{k+1} = BS_kA^T + B^TS_kA \tag{19}$$

A and B are the respective adjacency matrices of aug_A and aug_B, and S_0 is an all-ones matrix. So, the information on the connectivity similarity of the nodes is collected within S.

In our framework, we used the existing python implementation in graphsim,[6] which is based on NetworkX and call the algorithm `NodeSimilarityOpt`. To convert the similarity matrix S into a distance metric, we reduced it with the maximum linear sum assignment (lsa) to find the maximum node-node similarities as a set of similarity values. In the second step, we use the average node-node similarity from that result to compute the distance metric as follows:

$$dist_{NodeSim}(aug_A, aug_B) = 1 - \frac{\sum lsa(S)}{|lsa(S)|)} \tag{20}$$

Exas-Vectors. Lastly, we employed distance metrics from the code-clone domain using so-called Exas-Vectors to measure graph distances. Nguyen et al. [43] have shown that Exas-Vectors are able to reasonably approximate the GED, and thus they are eligible to measure the distance of AUGs. Exas-Vectors denote vectorizations of graphs whose elements represent the number of certain features present in the graphs. Their definition involves two different kinds of features: (p,q)-nodes and n-paths. (p,q)-nodes describe the individual nodes (e.g., denoted by their label function l_V) together with the number of incoming (i.e., p) and outgoing edges (i.e., q). N-paths describe paths of the size $length$ (i.e., number of visited nodes), where nodes and edges are identified via their respective label function (i.e., l_V and l_E). In our experiments, we omitted n-paths of size one (i.e., single nodes), since they are included in the (p,q)-node feature. Moreover, building on the results of Nguyen et al., we limited the maximum path length to four.

To measure the distance between two AUGs, we compute the norm of the difference between their respective Exas-vectors. Since the vectors may differ in their specific number and type of features, we first determined the shared features among the two vectors as sub-vectors containing only those features with their respective counts. Similarly, we also computed super-vectors containing a union of all features of both vectors, in which the respectively added features are filled up with zeros. We then computed two different norms, namely L1-norm and cosine-distance, on the Exas vectors. Since the cosine distance is less sensitive to individual differences in the feature count than the L1-norm, we also included the proportion of shared features for this distance. We refer to those distances as `ExasVectorL1Norm` and `ExasVectorCosine`. Assume vec_A and vec_B are the Exas vectors of the AUGs aug_A and aug_B, $\hat{vec}_A$ and $\hat{vec}_B$ are the respective super-vectors with all features, $\tilde{vec}_A$ and $\tilde{vec}_B$ the respective sub-vectors with all shared features, len a function to compute the length (i.e., number of elements) of a vector, and $maxVal$ a function to determine the maximum absolute value of a vector. Then, we can compute the respective distances as follows:

$$dist_{ExasVectorL1Norm}(aug_A, aug_B) = \\ ||\frac{\hat{vec}_A - \hat{vec}_B}{max(1, maxVal(\hat{vec}_A - \hat{vec}_B))}||_1 \tag{21}$$

$$dist_{ExasVectorCosine}(aug_A, aug_B) = \\ \lambda\frac{len(\tilde{vec}_A)}{len(vec_A)} + (1-\lambda)(1 - \frac{\langle \tilde{vec}_A, \tilde{vec}_B \rangle}{||\tilde{vec}_A||_2||\tilde{vec}_B||_2}), \lambda \in [0, 1] \tag{22}$$

$\langle \cdot, \cdot \rangle$ denotes the scalar product of two vectors. In our experiments we set $\lambda = 0.5$.

We also constructed two additional distance algorithms by splitting the AUGs into subgraphs. For this purpose, we cluster nodes of an AUG based on their related packages determined by the api function (cf. Section II-A). Then, we construct each subgraph by removing all nodes (together with their connected edges) from the original AUG that does not belong to that cluster. So, we obtain a list of subgraphs from a single AUG, each related to a certain API package and, regarding nodes for which no package could be determined, a special miscellaneous subgraph. Finally, we compute the distance between two AUGs by computing the distances of their subgraphs that belong to the same API package and averaging all resulting distances. During this mean computation, we ignore subgraphs that do not have a counterpart in the other AUG (i.e., those that depict a different API usage) as well as sub-distances that equal one. The rationale is that these values typically distort the distance computation with noise introduced by unrelated API usages. We reused the L1-norm and the cosine distance from above and refer to those subgraph distance algorithms as `ExasVectorSplitL1Norm` and `ExasVectorSplitCosine`, respectively.

[6] https://github.com/caesar0301/graphsim

IV. Experiment

Next, we describe our experiments and discuss their results.

A. Data and Experimental Setup

We analyzed the described distance algorithms based on two datasets. First, we used *MUBench*[7] by Amann et al. [6]. This dataset contains a set of fixed API misuses together with their repository information, the fixing commit, the fixed method, and more details. We selected all 116 misuse entries that are linked to a git repository, provide the fixing commit hash, as well as the containing method and source file path. Based on these entries, we generated AUGs of the respective misuse (i.e., commit before the fix) and the correct usage (i.e., commit after the fix) as well as the respective correction rule as described in our previous work [47]. Instead of manually determining the import statements to generate rules, we automatically included all external import statements, namely those not starting with the same prefix as the package of the analyzed source file. We could generate AUGs and corresponding correction rules for 96 of the 116 misuses (e.g., some source files could not be properly parsed).

The second dataset, *AU500*,[8] has been published by Kang et al. [26] and includes manually labeled correct API usages and API misuses. This dataset was constructed to provide an independent baseline to assess API-misuse detection tools. It comprises 500 API usages, 385 of which are correct usages, while the other 115 represent misuses. We were able to generate AUGs for 493 entries (114 misuses and 379 correct usages).

We transformed all generated AUGs and their correction rules into a dot-representation,[9] including their respective information on the API as well as the types and labels of nodes and edges. This way, we could conduct all of our experiments in the same python-based framework by converting and processing AUGs and correction rules with the NetworkX library.[10] Within our framework, we analyzed the *effectiveness* and the *efficiency* of the distance algorithms. For both, we first generated the correction rules from the MUBench dataset as reference usage in the form $aug_{rm} \rightarrow aug_{rc}$, where aug_{rm} represents the misuse part of the misuse and aug_{rc} its respective fix. Moreover, we denote the set of misuse AUGs as $M_{MUBench}$ and the set of correct usages as $C_{MUBench}$. For each rule, we then computed the analyzed distance function $dist_x$ to assess the rule's *applicability* as discussed in Section III-A. To this end, we computed the four individual distances $dist_x(aug_{rc}, aug_c)$, $dist_x(aug_{rc}, aug_m)$, $dist_x(aug_{rm}, aug_c)$, and $dist_x(aug_{rm}, aug_m)$, where $aug_c \in C_{MUBench}$ and $aug_m \in M_{MUBench}$. When computing these four distance values, we measured the time using python's `time` library.[11] For the algorithm `NetworkXGED`, we defined a timeout for individual computations, which we set to 15 seconds. We selected this value since it ensures that the maximum time to compute all four distances would be at one minute, which was comparable to the execution time of the other algorithms.

[7]https://github.com/stg-tud/MUBench

[8]https://github.com/ALP-active-miner/ALP

[9]https://graphviz.org/doc/info/lang.html

[10]https://networkx.org/

[11]https://docs.python.org/3/library/time.html

TABLE I
Number of *Applicable* rules

distance	#rules-sat	/	#rules
ExasVectorL1Norm	0	/	96
ExasVectorCosine	6	/	96
ExasVectorSplitL1Norm	2	/	96
ExasVectorSplitCosine	1	/	96
HungarianGED	0	/	96
NetworkXGED	15	/	96
HungarianMCS	0	/	96
NodeSimilarityOpt	0	/	92

Regarding *effectiveness*, we checked for all computed metrics whether each analyzed rule is *applicable* (i.e., satisfies Equations 3, 4, 5, and 6). We then counted the number of all satisfying rules per distance algorithm (i.e., on MUBench). To further assess a rule's ability to detect and fix misuses, we computed the distance values to AU500. More detailed, we checked each entry of AU500 for which the condition in Equation 2 holds, and thereby decided if this entry was identified as a misuse. For a reference rule $aug_{rm} \rightarrow aug_{rc}$, this means that an entry AUG aug_e in AU500 satisfies the condition $dist_x(aug_{rc}, aug_e) > dist_x(aug_{rm}, aug_e)$. Since the entries in AU500 are manually labeled as misuse or correct usage, we can compare this result against the labeled ground-truth and compute the rule's precision and recall. To assess the *efficiency*, we measured the execution time when computing the four distances for each entry on the MUBench dataset. We provide our experimental data, the framework, and all other scripts in our replication package.[1]

B. Effectiveness

In Table I, we summarize the number of *applicable* rules per distance algorithm. We can see that only the Exas vector algorithms and `NetworkXGED` algorithm found *applicable* rules. However, even in the best case (i.e., `NetworkXGED`), they found only a minority of 15 out of 96 possible rules. Note that for the `NodeSimilarityOpt`-algorithm, we could not compute the distance for four rules, and thus the number of checked rules is lower. Overall, we could identify 24 rules (20 unique ones) for four different metrics.

We then checked these 24 rules against the AU500 dataset using the respective algorithm to compute the distance. Based on the number of true and false positives, we computed the precision and recall, which we depict in Table II. Overall, we can see that for rules that detected at least one misuse (i.e., with $\#tp > 0$ or $\#fp > 0$), the precision and recall are constantly very low. In comparison to the results obtained by Kang et al. [26], who applied MUDetect (precision 27.6%, recall 29.6%) and ALP (precision 44.7%, recall 54.8%) using the same dataset, the simple distance metrics could not successfully discriminate misuses from correct usages. While we expected our rules to achieve a low recall, since they describe very specific fixes, they could not achieve the aspired high precision.

TABLE II
PRECISION AND RECALL OF APPLICABLE RULES TOGETHER WITH THEIR APPLIED DISTANCE METRIC AND THE NUMBER OF TRUE AND FALSE POSITIVES/NEGATIVES (I.E., #TP, #FP, #TN, #FN)

distance	rule_id	#fp	#tp	#fn	#tn	precision	recall
ExasVectorCosine	1_TuCanMobile	0	0	114	379	0.0%	0.0%
ExasVectorCosine	2_alibaba_druid	26	5	109	353	16.13%	4.39%
ExasVectorCosine	30_visualee	74	17	97	305	18.68%	14.91%
ExasVectorCosine	390_paho.mqtt.java	28	6	108	351	17.65%	5.26%
ExasVectorCosine	473_ntru	3	1	113	376	25.0%	0.88%
ExasVectorCosine	56_2_gora	2	3	111	377	60.0%	2.63%
ExasVectorSplitL1Norm	1_Apache_Commons_Math	42	5	109	337	10.64%	4.39%
ExasVectorSplitL1Norm	1_Mozilla_Rhino	0	0	114	379	0.0%	0.0%
ExasVectorSplitCosine	1_Mozilla_Rhino	0	0	114	379	0.0%	0.0%
NetworkXGED	1_Apache_Commons_Lang	52	14	100	327	21.21%	12.28%
NetworkXGED	1_Apache_Commons_Math	39	10	104	340	20.41%	8.77%
NetworkXGED	1_Closure_Compiler	16	3	111	363	15.79%	2.63%
NetworkXGED	1_Onosendai_-_A_Better_Deck	8	2	112	371	20.0%	1.75%
NetworkXGED	1_Screen_Notifications	3	4	110	376	57.14%	3.51%
NetworkXGED	1_WordPress_for_Android	94	20	94	285	17.54%	17.54%
NetworkXGED	29_visualee	41	9	105	338	18.0%	7.89%
NetworkXGED	2_Apache_Commons_Lang	47	9	105	332	16.07%	7.89%
NetworkXGED	2_Apache_Commons_Math	24	5	109	355	17.24%	4.39%
NetworkXGED	2_Closure_Compiler	24	6	108	355	20.0%	5.26%
NetworkXGED	2_alibaba_druid	31	11	103	348	26.19%	9.65%
NetworkXGED	30_visualee	61	11	103	318	15.28%	9.65%
NetworkXGED	361_Joda-Time	22	5	109	357	18.52%	4.39%
NetworkXGED	39_gae-java-mini-profiler	72	19	95	307	20.88%	16.67%
NetworkXGED	3_Closure_Compiler	23	5	109	356	17.86%	4.39%

TABLE III
MEAN/MEDIAN TIMES OF THE EFFICIENCY RESULTS ON MUBENCH

distance	time in sec (mean)	time in sec (median)
ExasVectorCosine	1.363347	0.046309
ExasVectorL1Norm	1.442757	0.049212
ExasVectorSplitCosine	1.163106	0.061127
ExasVectorSplitL1Norm	1.177104	0.062430
HungarianGED	0.010174	0.006016
HungarianMCS	0.007411	0.003391
NetworkXGED	15.552184	0.323910
NodeSimilarityOpt	0.043162	0.034601

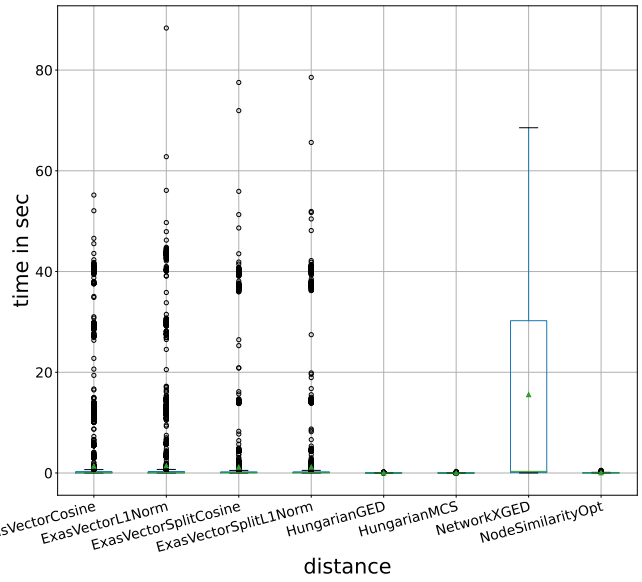

Fig. 3. Boxplot of the efficiency results on MUBench

C. Efficiency

Based on our time measurements, we obtained time distributions (cf. Figure 3) as well as median and mean times (cf. Table III) to compute the four mentioned distance values per rule. Based on our results, we observe that most algorithms require roughly one second except for `NetworkXGED`, which has a mean time of $\approx$ 15 seconds. This mean value is heavily influenced by a smaller number of long-lasting computations. Particularly, 1,841 of 5,467 distance computations last longer than 2 seconds. So, most algorithms can efficiently compute a distance except for `NetworkXGED`. Unfortunately, this algorithm was the one that found most *applicable* rules in our dataset (cf. Section IV-B).

D. Root Causes of Low Precision and Recall

Our results indicate that none of the algorithms we selected is sufficient (regarding precision and recall) to effectively discriminate correct usages from misuses. Thus, we conducted a qualitative, in-depth analysis of the root causes for the low precision and recall. In detail, for each algorithm, two reviewers analyzed a sample set of ten pairs of a correction rule $aug_{rm} \rightarrow aug_{rc}$ together with the complete misuse AUG aug_m and its respective corrected AUG aug_c originating from another misuse-correction pair (all from the MUBench dataset). For those samples, the conditions formulated in Equations 1 and 2 must hold. Moreover, we ensured that $rm \neq m$ and $rc \neq c$ to avoid that a correction rule is compared to itself. We were able to analyze such pairs for all metrics, but `HungarianGED` and

HungarianMCS. For those algorithms, we did not obtain any entry satisfying our previously mentioned conditions. Considering NetworkXGED, nine out of 10 entries have a correction rule and a misuse/correction AUG stemming from the same project. Therefore, we re-sampled this set and allowed only entries from different projects.

The two reviewers individually analyzed the samples to decide whether the usages contain at least one identical API-method call (agreement in 60 out of 70 cases), identify whether this method call is used in a similar context (agreement in 58 out of 70 cases), and add a textual discussion justifying their decision. For instance, assume a misuse of an unlabeled button, which was fixed by a correction rule adding a textual label. Further, a similar correction fixed the same issue by adding an image to the empty button. Then, the reviewer comparing these entries would decide that the same API method call is used, for example, the constructor of the button, and the correction is done in the same context, namely labeling a button. Finally, the reviewers discussed their decisions with each other and identified the root causes of falsely matched entries. Next, we discuss general and metric-specific causes for false misuse detection. For each cause, we formulate hypotheses, which have to be evaluated in subsequent research.

General. We observed that almost all metrics tend to work better when comparing API usages in the same project than with an external usage. This seems to be reasonable, since an API usage may share more structural similarities to a usage in the same project than to an external usage. Thus, *applicable* rules may hardly apply to external projects:

> Hypothesis 1
>
> API misuse detection with correction rules using distance metrics will be more precise if it is applied within the same project than on an external one.

GED. We obtained most results for the NetworkXGED algorithm. This algorithm had issues with the large number of order edges in AUGs. Particularly, many order-edges are "recycled" without any further costs in the GED computation. Therefore, the differences in the node labels, which we perceived as more important for detecting API misuses, are under-represented. For this reason, we argue that:

> Hypothesis 2
>
> API misuse detection with correction rules using the GED is more precise if the costs are adapted to edits of certain AUG element types.

MCS. We obtained no rules satisfying the sampling condition, and thus we conclude that computing graph distances using MCSs may not be a valid metric for detecting API misuses.

Node-Node Similarity. In our analysis, we noticed that this algorithm usually computes a low distance (i.e., high similarity) in case the graphs share nodes that are similarly connected to each other; even though the respective node labels differ. For instance, we observed many matches that were caused by handling an exception even though the exception and its causing API differ. This happens since nodes, which handle an exception, tend to share structural similarities (e.g., catch-blocks, initialization of an exception object). Based on this insight, we hypothesize:

> Hypothesis 3
>
> API misuse detection with correction rules using the Node-Node Similarity is more precise when including the similarity of the node labels in the computation.

Exas Vectors. We determined two issues regarding the Exas-vector algorithms. First, in some cases, the similarity originates from trivial and individual features that match, such as (p,q)-nodes of <return> or <throw> nodes. This matches many non-similar API usages (e.g., since many API usages contain <return>-nodes), which is why:

> Hypothesis 4
>
> API misuse detection with correction rules using the Exas Vectors is more precise when including only features from the vector containing relevant API information.

Second, especially for the ExasVectorL1Norm algorithm, we found many cases in which a high frequency of a single feature diminishes the effect of other differences of the two vectors. Particularly, assume two super-vectors $v_1 = (1, 0, 1)$ and $v_2 = (0, 1, 0)$ from their respective Exas vectors in which the position in the vector denotes a unique feature. Then, $dist_{ExasVectorL1Norm}(v_1, v_2) = 1$ (i.e., maximum distance), which is reasonable since both vectors share no feature. However, assume $v_2' = (0, 2, 0)$, then $dist_{ExasVectorL1Norm}(v_1, v_2') = \frac{2}{3}$. In fact, the normalization of the subtraction $v_1 - v_2'$ causes a decreasing distance value even though both vectors still do not share any feature. To avoid normalization, one may substitute the frequencies with simple indicators (i.e., 0 if the feature is absent and else 1), so:

> Hypothesis 5
>
> API misuse detection with correction rules using the ExasVectorL1Norm metric is more precise when using Exas vectors with indicators rather than frequencies.

Finally, we investigated the effect of splitting AUGs into API-specific sub-graphs. We found that splitting usually improves the results, while the corresponding metrics still suffer from the previously mentioned problems. As a consequence, we argue:

> Hypothesis 6
>
> API misuse detection with correction rules using the Exas vectors is more precise when splitting graphs into API-specific subgraphs.

V. Threats to Validity

Following the classification by Siegmund et al. [60], we consider threats to the *internal* and *external* validity.

A. Internal Validity

Threats to the internal validity describe phenomena of our setup that may harm the trustfulness of the results. First, our implementation may contain errors that eventually lead to null results. While we cross-checked and tested our implementation on sample data, we still cannot ensure its correctness. Moreover,

conceptional issues with the algorithms can taint the results (e.g., different methods produced by overloading could result in the same features in Exas vectors). Thus, for transparency and replication, we publish our code.[1] Second, whether a misuse in the original dataset represents a real misuse is dependent on the original decision of the respective authors. Since we did not re-check the validity of the ground truth, this may influence the results. Third, even though we carefully reviewed existing literature, we have arguably not identified, nor used, all existing algorithms—which may perform better than the analyzed ones. However, this could be easily fixed by integrating and comparing other algorithms in our published framework. Finally, even though the reviewers individually performed the root-cause analysis and discussed their decisions with each other, it is still a subjective view. Other reviewers may find different issues or argue that our derived hypotheses are not reasonable. Thus, we recommend further experiments to validate our hypotheses, and potentially derive new ones.

B. External Validity

External validity considers phenomena that may prevent us from generalizing our results. In our experiments, we restricted the applicability of the algorithms on API usage graphs, and thus on the Java programming language. While we expect that AUGs can be adapted to other programming languages, we cannot ensure that the results apply to other languages. Moreover, we cannot state whether the results apply to other graph types, such as control-flow graphs or the extended AUGs introduced by Kang et al. [26]. This will be subject of further research. Finally, the MUBench dataset may not be a representative set of API misuses, which could cause their limited applicability to detect misuses in the AU500 dataset. Thus, other datasets should be researched, such as the ones provided by Kechagia et al. [27] or ourselves [46].

VI. Related Work

Our work relates to surveys on code similarity techniques, graph similarity, and comparative studies of automated software-engineering techniques.

A. Surveys on Code Similarity

Code similarity is actively used for code-clone detection [30], [55], analyzing plagiarism and software license compliance [51], code search [9], [24], [56], code quality analysis [51], as well as automated program repair (e.g., plastic surgery hypothesis) [34]. Most prominently, Haq et al. [23] surveyed 70 techniques on binary code similarity regarding their applications, characteristics, implementation, benchmarks, and evaluation techniques. They focus on techniques for binary code, and thus deal with additional noise, for instance, by different compiler settings or obfuscation. Also, Haq et al. did not perform comparative experiments. Ragkhitwetsagul et al. [51] analyzed 30 different code similarity techniques regarding their applicability to certain degrees of code changes, for instance, global and local changes. The study focuses on detecting copied and modified code, while our study refers to detecting similarities between correct and false API usages. Moreover, our study involves AUGs as intermediate code representation, while Ragkhitwetsagul et al. used text similarity and simpler code structures, such as ASTs.

B. Surveys on Graph Similarity

While driven by the respective domains in which graph-similarity algorithms are needed, some general surveys on graph similarity exist. For instance, Aggarwal [2] discusses the general problem of (sub-)graph isomorphism and graph distance together with state-of-the-art algorithms in his data mining book (cf. Chapter 17). Gao et al. [20] provide a survey on algorithms computing the graph edit distance. We used these works as sources to systematically select candidate distance algorithms. Ren et al. [52] observed that the performance of subgraph isomorphism algorithms is correlated to the structure of graphs. Thus, we directly analyzed the performance of realistic AUGs and the respective AUG correction rules. Other surveys consider more advanced metrics, such as parallel algorithms to compute graph distances [29] or deep learning techniques [37]. We did not consider these algorithms, since they require more complex computation and hardware as well as carefully selected training sets. However, in case our hypotheses are shown to not be true, will not significantly improve the results, or suffer from poor performance, we will analyze more complex algorithms.

C. Studies on Automated Software-Engineering Techniques

Our work aligns with different comparative studies on techniques in the automated software-engineering domain, such as automated program repair in general [17], automated repair of API misuses [27], static code analysis for detecting security vulnerabilities [22], fault localization techniques [69], or the performance of API misuse detectors [4]. To the best of our knowledge, no previous work exists that evaluates graph-distance algorithms for API usages comparisons. So, even though our results did not indicate a clear benefit of such algorithms, we still contribute to the existing body of knowledge.

VII. Conclusion

Using APIs is accompanied by their potential misuses causing negative effects in the implemented software. Thus, researchers developed automated techniques to detect such misuses. For this purpose, recent techniques represent API usages as AUGs and compare usages to known correct usages or misuses to detect and ideally repair misuses. However, to the best of our knowledge, no studies analyzing existing graph-distance algorithms to detect API misuses exist. Therefore, in this paper, we formalized this problem and the desired goal, introduced a set of well-known algorithms, and conducted a study using two independent data sets of real API misuses. Our results indicate that the analyzed algorithms fail to effectively discriminate correct API usages from misuses. Based on a post-analysis, we identified the potential issues of individual algorithms and derived six hypotheses for further improvements. In the future, we will analyze these hypotheses and investigate more advanced techniques for graph distance computation.

References

[1] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, "An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems," in *Proceedings of the 4th International Conference on Pattern Recognition Applications and Methods (ICPRAM)*. SciTePress, 2015.

[2] C. C. Aggarwal, *Mining Graph Data*. Cham: Springer International Publishing, 2015, pp. 557–587.

[3] M. Allamanis and C. Sutton, "Mining Idioms from Source Code," in *Proceedings of the 22nd International Symposium Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 472–483.

[4] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A Systematic Evaluation of Static API-Misuse Detectors," *IEEE Transactions on Software Engineering (TSE)*, vol. 45, no. 12, pp. 1170–1188, 2019.

[5] S. Amann, "A Systematic Approach to Benchmark and Improve Automated Static Detection of Java-API Misuses," Ph.D. dissertation, Technische Universität Darmstadt, Jun. 2018.

[6] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "MUBench: A Benchmark for API-Misuse Detectors," in *Proceedings of the 13th International Workshop on Mining Software Repositories (MSR)*. ACM, 2016.

[7] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating Next Steps in Static API-Misuse Detection," in *Proceedings of the 16th International Workshop on Mining Software Repositories (MSR)*. IEEE, 2019.

[8] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proceedings of the 29th Symposium on Principles of Programming Languages (POPL)*. ACM, 2002, pp. 4–16.

[9] M. H. Asyrofi, F. Thung, D. Lo, and L. Jiang, "AUSearch: Accurate API Usage Search in GitHub Repositories with Type Resolution," in *Proceedings of the 27th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2020, pp. 637–641.

[10] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. V. Dooren, "A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching," *SIAM Review*, vol. 46, no. 4, p. 647–666, Apr. 2004.

[11] J. Bosch and P. Bosch-Sijtsema, "From Integration to Composition: On the Impact of Software Product Lines, Global Development and Ecosystems," *Elsevier Journal of Systems and Software (JSS)*, vol. 83, no. 1, pp. 67–76, 2010.

[12] H. Bunke, "On a relation between graph edit distance and maximum common subgraph," *Elsevier Pattern Recognition Letters (PRLEDG)*, vol. 18, no. 8, pp. 689–694, 1997.

[13] H. Bunke and X. Jiang, *Graph Matching and Similarity*. Boston, MA: Springer, 2000, pp. 281–304.

[14] H. Bunke and K. Shearer, "A graph distance metric based on the maximal common subgraph," *Elsevier Pattern Recognition Letters (PRLEDG)*, vol. 19, no. 3, pp. 255–259, 1998.

[15] C. Chen, Z. Xing, Y. Liu, and K. L. X. Ong, "Mining Likely Analogical APIs across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding," *IEEE Transactions on Software Engineering (TSE)*, pp. 1–1, 2019.

[16] D. F. Crouse, "On implementing 2d rectangular assignment algorithms," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 52, no. 4, pp. 1679–1696, 2016.

[17] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts," in *Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 302–313.

[18] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Elsevier Science of Computer Programming (SCP)*, vol. 69, no. 1, pp. 35 – 45, 2007, Special Issue on Experimental Software and Toolkits.

[19] M. Gabel and Z. Su, "Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces," in *Proceedings of the 16th International Symposium Foundations of Software Engineering (FSE)*. ACM, 2008, pp. 339–349.

[20] X. Gao, B. Xiao, D. Tao, and X. Li, "A Survey of Graph Edit Distance," *Springer Pattern Analysis and Applications*, vol. 13, no. 1, pp. 113–129, 2010.

[21] P. L. Gorski, Y. Acar, L. Lo Iacono, and S. Fahl, "Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs," in *Proceedings of the 38th International Conference on Human Factors in Computing Systems (CHI)*. ACM, 2020, p. 1–13.

[22] K. Goseva-Popstojanova and A. Perhinschi, "On the Capability of Static Code Analysis to Detect Security Vulnerabilities," *Elsevier Journal of Information and Software Technology (IST)*, vol. 68, pp. 18–33, 2015.

[23] I. U. Haq and J. Caballero, "A Survey of Binary Code Similarity," *ACM Computing Surveys*, vol. 54, no. 3, Apr. 2021.

[24] R. Holmes and G. C. Murphy, "Using Structural Context to Recommend Source Code Examples," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. ACM, 2005, pp. 117–125.

[25] D. Hou and L. Li, "Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions," in *Proceedings of the 19th International Conference on Program Comprehension (ICPC)*. IEEE, 2011, pp. 91–100.

[26] H. J. Kang and D. Lo, "Active Learning of Discriminative Subgraph Patterns for API Misuse Detection," *IEEE Transactions on Software Engineering (TSE)*, pp. 1–1, 2021.

[27] M. Kechagia, S. Mechtaev, F. Sarro, and M. Harman, "Evaluating Automatic Program Repair Capabilities to Repair API Misuses," *IEEE Transactions on Software Engineering (TSE)*, pp. 1–1, 2021.

[28] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," in *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1998, p. 668–677.

[29] G. Kollias, M. Sathe, O. Schenk, and A. Grama, "Fast parallel Algorithms for Graph Similarity and Matching," *Elsevier Journal of Parallel and Distributed Computing*, vol. 74, no. 5, pp. 2400–2410, 2014.

[30] R. Koschke, "Survey of Research on Software Clones," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no. 06301. IBFI, 2007.

[31] C. W. Krueger, "Software Reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[32] J. Krüger and T. Berger, "An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, pp. 432–444.

[33] M. Lamothe and W. Shang, "When APIs Are Intentionally Bypassed: An Exploratory Study of API Workarounds," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020, p. 912–924.

[34] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated Program Repair," *Communications of the ACM*, vol. 62, no. 12, p. 56–65, Nov. 2019.

[35] C. Le Goues and W. Weimer, "Measuring Code Quality to Improve Specification Mining," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 175–190, Jan 2012.

[36] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," in *Proceedings of the 5th Joint Meeting of the European Software Engineering Conference/-Foundations of Software Engineering (ESEC/FSE)*, vol. 30, no. 5. ACM, 2005, pp. 296–305.

[37] G. Ma, N. K. Ahmed, T. L. Willke, and S. Y. Philip, "Deep Graph Similarity Learning: A Survey," *Springer Data Mining and Knowledge Discovery*, pp. 1–38, 2021.

[38] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How Can I Use This Method?" in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 880–890.

[39] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, 1957.

[40] V. Murali, S. Chaudhuri, and C. Jermaine, "Bayesian Specification Learning for Finding API Usage Errors," in *Proceedings of the11th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 151–162.

[41] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through Hoops: Why Do Java Developers Struggle with Cryptography APIs?" in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, 2016, p. 935–946.

[42] D. Nam, A. Horvath, A. Macvean, B. Myers, and B. Vasilescu, "MARBLE: Mining for Boilerplate Code to Identify API Usability Problems," in *Proceedings of the 34th International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, p. 615–627.

[43] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2009, pp. 440–455.

[44] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring API Embedding for API Usages and Applications," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, p. 438–449.

[45] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based Mining of Multiple Object Usage Patterns," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 2009, pp. 383–392.

[46] S. Nielebock, P. Blockhaus, J. Krüger, and F. Ortmeier, "AndroidCompass: A Dataset of Android Compatibility Checks in Code Repositories," in *Proceedings of the 18th International Workshop on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 535–539.

[47] S. Nielebock, R. Heumüller, J. Krüger, and F. Ortmeier, "Cooperative API Misuse Detection Using Correction Rules," in *Proceedings of the 42nd International Conference on Software Engineering - New Ideas and Emerging Results (ICSE-NIER)*. ACM, 2020, p. 73–76.

[48] S. Nielebock, R. Heumüller, K. M. Schott, and F. Ortmeier, "Guided Pattern Mining for API Misuse Detection by Change-Based Code Analysis," *Springer Automated Software Engineering (ASE)*, vol. 28, no. 15, pp. 1–48, 2021.

[49] D. S. Oliveira, T. Lin, M. S. Rahman, R. Akefirad, D. Ellis, E. Perez, R. Bobhate, L. A. DeLong, J. Cappos, and Y. Brun, "API Blindspots: Why Experienced Developers Write Vulnerable Code," in *Proceedings of the 14th Symposium on Usable Privacy and Security (SOUPS)*. USENIX, 2018, pp. 315–328.

[50] N. Patnaik, J. Hallett, and A. Rashid, "Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries," in *Proceedings of the 15th Symposium on Usable Privacy and Security (SOUPS)*. USENIX, 2019.

[51] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A Comparison of Code Similarity Analysers," *Springer Empirical Software Engineering (EMSE)*, vol. 23, no. 4, pp. 2464–2519, 2018.

[52] X. Ren, J. Wang, N. Franciscus, and B. Stantic, "Experimental Clarification of some Issues in Subgraph Isomorphism Algorithms," in *Asian Conference on Intelligent Information and Database Systems*. Springer, 2018, pp. 71–80.

[53] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE Software*, vol. 26, no. 6, pp. 27–34, 2009.

[54] M. P. Robillard and R. DeLine, "A field Study of API Learning Obstacles," *Springer Empirical Software Engineering (EMSE)*, vol. 16, no. 6, pp. 703–732, 2011.

[55] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A qualitative Approach," *Elsevier Science of Computer Programming (SCP)*, vol. 74, no. 7, pp. 470–495, 2009.

[56] N. Sahavechaphan and K. Claypool, "XSnippet: Mining For Sample Code," in *Proceedings of the 21st Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2006, pp. 413–430.

[57] A. Sanfeliu and K.-S. Fu, "A distance measure between attributed relational graphs for pattern recognition," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 3, pp. 353–362, 1983.

[58] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird, "Recommending Random Walks," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 2007, pp. 15–24.

[59] F. Serratosa, "Graph edit distance: Restrictions to be a metric," *Elsevier Pattern Recognition*, vol. 90, 06 2019.

[60] J. Siegmund, N. Siegmund, and S. Apel, "Views on Internal and External Validity in Empirical Software Engineering," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 9–19.

[61] M. Stauffer, T. Tschachtli, A. Fischer, and K. Riesen, "A Survey on Applications of Bipartite Graph Edit Distance," in *Proceedings of the 10th International Workshop on Graph-Based Representations in Pattern Recognition*. Springer, 2017.

[62] S. Thummalapenta and T. Xie, "Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web," in *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 204–213.

[63] C. Treude and M. P. Robillard, "Augmenting API Documentation with Insights from Stack Overflow," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 392–403.

[64] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects," in *Proceedings of the 36th International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 35–45.

[65] A. Wasylkowski and A. Zeller, "Mining Temporal Specifications from Object Usage," *Springer Automated Software Engineering (ASE)*, vol. 18, no. 3-4, pp. 263–292, 2011.

[66] T. Zhang, B. Hartmann, M. Kim, and E. L. Glassman, "Enabling Data-Driven API Design with Community Usage Data: A Need-Finding Study," in *Proceedings of the 38thInternational Conference on Human Factors in Computing Systems (CHI)*. ACM, 2020, p. 1–13.

[67] H. Zhong and Z. Su, "An Empirical Study on Real Bug Fixes," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 913–923.

[68] C. Ziftci and J. Reardon, "Who broke the build? Automatically identifying changes that induce test failures in continuous integration at Google Scale," in *Proceedings of the 39th International Conference on Software Engineering - Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2017, pp. 113–122.

[69] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An Empirical Study of Fault Localization Families and Their Combinations," *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 2, pp. 332–347, 2021.