IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 1, 2019, accepted October 29, 2019, date of publication November 5, 2019, date of current version November 19, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2951637

User Interface Code Retrieval: A Novel Visual-Representation-Aware Approach

YINGTAO XIE[1,2], TAO LIN[1], AND HONGYAN XU[1]
[1]College of Computer Science, Sichuan University, Chengdu 610017, China
[2]Education and Information Technology Center, China West Normal University, Nanchong 637002, China

Corresponding author: Tao Lin (lintao@scu.edu.cn)

This work was supported in part by the Special Funds for High-Tech Industries Development of Nanchong Science and Technology Bureau, China, under Grant 18YFZJ0022, in part by the Meritocracy Research Funds of China West Normal University, China, under Grant 17YC188, and in part by the Innovation Team Project of China West Normal University, China, under Grant CXTD2017-5.

ABSTRACT Code reuse has been perceived to be an effective tool for user interface development that is known to be complex and messy. A fundamental problem in user interface code reuse is how to effectively retrieve working code from existing code repositories, which renders functional interfaces similar to a programmer's input sketch. Existing generic code retrieval techniques are normally inadequate for user interface code due to its associated visual information. In this paper, we put forward a novel approach that retrieves matching candidate code of an interface sketch based on their visual-representation trees. To better capture sub-region similarities, we propose to decompose a visual-representation tree into an ordered list of paths (or sequences). We then devise a novel algorithm to measure the similarity between two sequences by leveraging various visual information and another algorithm to calculate the similarity of two ordered lists by considering the relative positions of different widgets. To evaluate our solution, we carefully design an experiment to generate the similarity scores of 6,750 user interface pairs based on real users' judgement. Based on this ground truth, we show that our method is able to generate accurate similarity scores, outperforming several state-of-the-art competitors.

INDEX TERMS User interface code retrieval, visual-representation tree, tree matching, sequence matching.

I. INTRODUCTION

Code reuse refers to the concept of leveraging existing sections of code, templates, functions, and procedures to build new software. With the upsurge in the open-source software movement, code reuse has been an important and active research area, and various code reuse techniques have been proven to be effective tools to speed up software development and/or improve code quality. Code reuse is particularly useful for user interface development, which is indispensable to today's applications.

User interface development is known to be complex and messy [1], [2]. It typically involves understanding complicated widgets, building multiple prototypes, realizing dynamic user interactions and generating enormous test cases [3]. The resultant code is often redundant, bug-ridden, and difficult to maintain. It is thus desirable to have a powerful (semi-)automated tool for programmers to efficiently generate high-quality functional user interfaces from a sketch (e.g., a picture, an SVG file or an XUL file) by making use of the existing code repositories (e.g., GitHub and Black Duck Open Hub), which contain a huge amount of lines of well-maintained code. The key challenge here is how to retrieve working code that renders similar interfaces so that the programmers can easily judge its usefulness and make slight modifications to match the sketch.

There has been some research on code retrieval in more general cases [4]–[6]. Unfortunately, such techniques do not fit user interface code well. This is because user interface code, compared with generic code, has the following unique characteristics. First, user interface code can normally be represented by a tree structure, which encodes the *visual* hierarchical relationship of UI components (see Figure 1 for an example of a user interface, its code snippet and the corresponding tree structure). We call such a tree structure a *visual-representation* tree. In contrast, while it is possible to extract some tree structures, for example, the abstract syntax tree (AST) [7], from generic code, they encode code's *semantic*

The associate editor coordinating the review of this manuscript and approving it for publication was Eunil Park.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

FIGURE 1. A user interface, its corresponding code and the visual-representation tree. In the visual-representation tree, the "V" label of an internal node indicates that its children are organized in a vertical layout, and the numbers in the leaf nodes indicate the indexes of the widgets they represent.

structure, not its visual representation information. Second, user interface code is written to render a visual user interface. Intuitively the usability of returned candidate code is often judged based on its visual representation by a programmer, without knowing the exact underlying code logic. Making use of this fact can largely improve the performance of user interface code retrieval.

To our best knowledge, the only method tailored to user interface code retrieval is the ad-hoc rule-based method [1], [2]. The general idea is to first collect candidate code by using a set of keywords that are provided by users and then validate the candidates by a rule-based tree matching algorithm. There are at least two observable limitations of this approach. First, while keywords provide an effective way to narrow down the search space, identifying the appropriate keywords requires much trial and error, and keyword-based retrieval generally cannot provide the fine-granularity capability to examine candidate code. Second, the rules used are ad-hoc in essence (e.g., every user-specified component has to be matched with a UI widget). The resulting scores are not well justified and do not necessarily reflect one's intuition. Our experiments also confirm that this ad-hoc method is often unable to achieve reasonable performance in practice. In this paper, we put forward a novel algorithm for effective user interface code retrieval by addressing the two aforementioned unique characteristics. Our insight is to match candidate code with the sketch *directly* based on their visual-representation trees, which effectively bypasses the bottleneck of keyword-based search. However, designing an effective tree matching algorithm that takes into consideration the visual representation is a challenging task for several reasons. First, the final visual effects and tree structures often do not have "one-to-one correspondence". Consider the examples given in Figure 2. The tree structures of the

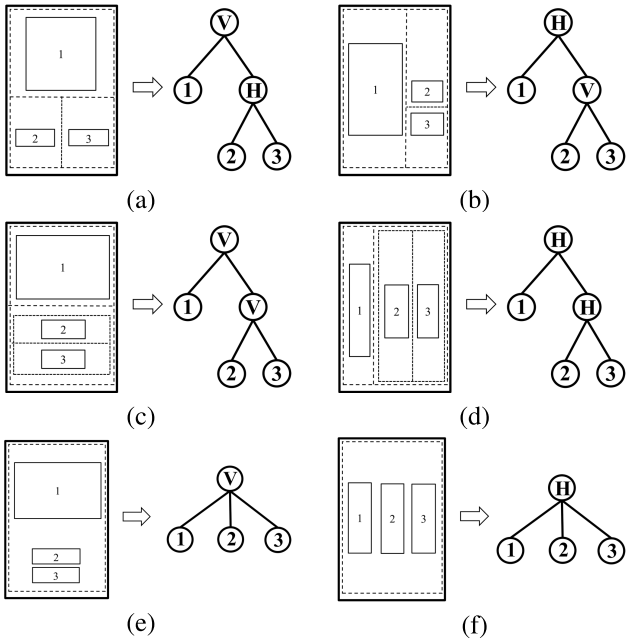

FIGURE 2. Several sample user interfaces and their visual-representation trees, where "V" indicates a vertical layout and "H" a horizontal layout. For example, in (a) the user interface is vertically divided into the upper region containing widget 1 and the lower region containing widgets 2 and 3; the lower region has a horizontal layout between widgets 2 and 3.

four interfaces in Figure 2(a)-(d) are identical, but their layouts are substantially different. In contrast, the tree structures of the interfaces in Figure 2(c) and (e) or the ones in Figure 2(d) and (f) are very different, but the visual effects are similar. Second, user interface code retrieval often does not expect a *global* match, which is an unlikely event in reality.

A user might provide an incomplete sketch or an initial idea for code retrieval. Having a highly similar and reusable sub-region is also desirable for code reuse. Third, not only the tree structure of user interface code but also the attributes of tree nodes play an important role in determining the similarity between the sketch and candidate code. For example, the relative sizes of different UI widgets largely affect a user's perception of a user interface. For these reasons, traditional tree matching algorithms [8]–[10] are generally not suitable for our purpose.

The focus of this paper is then to design a novel tree matching algorithm that considers both the tree structure and the visual representation information for user interface code retrieval. We summarize our key technical contributions as follows.

- We put forward a novel visual-representation-aware solution for user interface code retrieval, which well addresses the peculiarities of user interface code. To bypass the bottleneck of keyword-based code search, as used in the existing research, we propose to use a visual-representation tree based approach.
- We propose to decompose a visual-representation tree into an ordered list of paths, which gives the flexibility of identifying similar sub-regions and better efficiency. We then devise a novel algorithm to compare the similarity of two sequences by leveraging various visual information and another algorithm to calculate the similarity between two ordered lists of paths, which takes into consideration the relative positions of different UI widgets in the original interfaces.
- Since there is no standard dataset available for evaluating user interface code retrieval, we perform a carefully designed experiment to collect a ground truth test set, which requires non-trivial efforts. We make this dataset public to facilitate the research on user interface code retrieval and relevant fields. Based on the ground truth, we compare our method with a few state-of-the-art methods, including the only existing method for user interface code retrieval, and show the superiority of our solution.

The rest of our paper is organized as follows. We provide a literature review in Section II. In Section III, we discuss the details of our solution. Section IV presents an experimental study to demonstrate the effectiveness of our approach. Finally, we conclude the paper in Section V.

II. RELATED WORK

In this section, we review the literature in several relevant fields, including code retrieval, tree matching and sequence matching.

A. CODE RETRIEVAL

Code retrieval is one of the fundamental directions in code reuse research. Most studies focus on generic code retrieval. Early works [11], [12] typically adopt keyword-based approaches, which retrieve candidate code by matching the input keywords. In view of the limitations of keyword-based methods, Ye and Fischer [13], [14] present a system called CodeBroker that uses developers' partially written programs as implicit queries for code retrieval. Chien and Chien [15] propose to retrieve software code by using natural language processing (NLP) to identify the variation points (VPs) and components from software product lines. The similarity of code is then computed by weighting the similarity of VPs and components. Wang *et al*. [16] develop a specialized graph query language that allows users to specify both free-form topics and complex dependence relations. Based on this language, a semantic dependence search engine is designed to retrieve code snippets which are represented by System Dependence Graphs (SDGs).

In recent years, code retrieval has been widely used in code clone detection, an important problem for software maintenance and evolution. Roy and Cordy [17] categorize the existing works on code clone detection into four clone types based on the extent of code similarity. The first three types mainly focus on textual similarity whereas the last type considers functional similarity. Some representative works along this line include [4], [18]–[21]. Ducasse *et al.* [18] propose a language independent approach based on a simple string matching method which directly compares source code. Li *et al.* [19] develop a token-based code clone detection tool called CP-Miner, which treats each line of the source code as a collection of segments to construct a sequence and utilizes frequent subsequence mining techniques to detect duplicate segments. Jiang *et al.* [20] present an efficient tree similarity algorithm called DECKARD. It represents source code as trees which can be subdivided into subtrees with numerical vectors and clusters these vectors to compute similarity. Chen *et al.* [21] make use of a geometry characteristic of dependency graphs to measure the similarity between code fragments of two different applications. White *et al.* [4] propose to use recurrent neural networks (RNNs) to encode source code in the lexical level and construct abstract syntax trees (ASTs) in the syntactic level, and combine these two levels of information to detect code clone.

All the above methods are designed for generic code and in general do not address the particularities of user interface code well. In contrast, our solution is tailored to user interface code retrieval with an emphasis on the visual effects rendered by code.

B. TREE MATCHING

Since we propose to match candidate code with a sketch by comparing their visual-representation trees, we review existing tree matching algorithms. There have been some well-established *general-purpose* tree matching algorithms. Zhang and Shasha [8] consider the problem of calculating the similarity of two trees by the edit distance. They propose a simple dynamic programming algorithm which achieves better time and space complexity. The idea is to consider the distance between two ordered forests and carefully eliminate

certain subtree-to-subtree distance calculations. Another line of research studies the tree inclusion problem: Given two ordered labelled trees S and T, how can S be obtained from T by deleting nodes? Kilpeläinen and Mannila [9] develop the idea of left embeddings as the basic algorithm and further improve it by maintaining a table which stores intermediate computation results. Bille and Gørtz [10] present an algorithm that improves the best known time complexities with only linear space. The main idea is to construct a data structure on a tree supporting a small number of procedures on subsets of nodes of the tree. All such general-purpose algorithms do not consider the visual effects associated with a visual-representation tree and thus cannot provide similarity scores that are consistent with programmers' perception. In Section IV, we experimentally demonstrate the superiority of our solution over a general-purpose algorithm.

To our best knowledge, the only tree matching algorithm customized for user interface code is the rule-based method [1], [2]. However, as explained before, this algorithm is ad-hoc and does not work well in practice.

C. SEQUENCE MATCHING

An important piece of our solution is pairwise comparison of tree paths. A tree path is essentially a sequence. A variety of classic sequence (or string) matching metrics have been proposed, including the Euler distance, Manhattan distance, Markov distance, and Chebyshev distance. We refer the interested readers to Chapter 2 in [22] for more details of these metrics. The similarity of sequences can also be calculated using the Pearson correlation coefficient or cotangent similarity. Later, Levenshtein [23] proposes the Levenshtein distance method for strings. It measures the difference of two strings by calculating the minimum number of single-character edits (i.e., insertions, deletions or substitutions) required to transfer a string to another string.

Another line of studies makes use of features extracted from sequences to compare their similarity. Hirschberg [24] proposes to use the *longest common subsequence* (LCS) of two sequences to compute the similarity. In a more recent study, Tsai [25] researches a variant of the LCS problem called the constrained longest common subsequence problem. Given strings S_1, S_2 and P, the constrained longest common subsequence problem for S_1 and S_2 with respect to P is to find the longest common subsequence *lcs* of S_1 and S_2 such that P is a subsequence of *lcs*. Wang [26] argues that the LCS measure ignores information contained in the second, third, ..., longest subsequences and suggests to use the number of *all common subsequences* (ACSs) as a measure of sequence similarity. Yang *et al.* [27] studies the *multiple longest common subsequence* (MLCS) problem which aims to find the longest common subsequences of 3 or more sequences. They formulate the MLCS problem into a graph search problem and present two space-efficient anytime MLCS algorithms.

Lin *et al.* [28] propose to represent trees as multidimensional sequences and measure their similarity on the basis of their sequence representations. Different sequence similarity measures, including all common subsequences, the longest common subsequence and dynamic time warping, are combined to evaluate tree similarity. While sharing a similar idea of representing a tree by multiple sequences, our solution differs from this approach in a fundamental way. This approach is developed to purely compute the structure similarity of trees. In contrast, our method distinguishes itself by novel techniques to quantify the similarity in terms of visual representation. None of the aforementioned sequence structure similarity measures are used in our solution. We do not see any feasible way to extend the approach in [28] to handle visual information. We consider this approach as a competitor in our experimental evaluation. As suggested by the experimental results, considering visual information is key to user interface code retrieval.

Similarly, all existing sequence matching algorithms focus on merely *structural matching* and ignore the visual representation implied by such structures. In this paper, we present a new sequence matching algorithm that fully leverages visual representation information, which is more effective for user interface code, as shown in our experiments.

III. OUR USER INTERFACE CODE RETRIEVAL SOLUTION

In the section, we present our novel visual-representation-aware user interface code retrieval solution. As explained before, the crux boils down to an effective tree matching algorithm tailored to user interface code. In what follows, we describe our solution in detail.

A. CREATE SEQUENCE REPRESENTATION

The first step of our solution is to convert an input sketch and candidate code into visual-representation trees that encode sufficient visual information for subsequent comparisons. While this is not our paper's focus, we discuss it below for the reason of completeness.

To convert an input sketch into a visual-representation tree, we adapt the *pix2code* model [29] to automatically extract its code structure. To train our model, it first requires a large number of labeled sketches and the corresponding user interface code. In our case, we crawl GitHub to obtain user interface code and generate their user interfaces as sketches. We manually label the types and positions of UI elements in each sketch. We can then train a CNN model (i.e., VGGNet) to learn the types and positions of UI elements from a given sketch. The output of the CNN model only indicates the leaf nodes of the visual-representation tree. We need to train another seq2seq network to recover the internal nodes of the visual-representation tree. We first construct a node sequence from the output of the CNN model based on their positions. For this sketch's code, we construct a visual-representation tree and generate its node sequence using depth-first search. These two sequences will be fed into the *seq2seq* network as input and output, respectively. By using these two models, we can generate the visual-representation tree from a sketch. We note that while

this approach is able to generate a high-quality tree for our purpose, it is, unfortunately, not sufficient to produce either fully functional code or production-level code. For example, it can neither leverage various UI design patterns nor pass the various types of testings required for production [3].

Converting candidate code into a visual-representation tree is relatively easy. We adapt existing Python libraries to parse candidate code and build the corresponding visual-representation tree.

The nodes of a visual-representation tree fall into two categories: A leaf node represents a widget (e.g., a button, a seek bar or a text view) in the interface. A non-leaf node does not correspond to a particular widget and is not directly visible to users. It can be viewed as a container which contains widgets or other containers. It normally controls the layout orientation of its children.

To capture adequate visual information in a visual-representation tree, we extract some visual-related attributes and record them in the nodes. For a non-leaf node, we record the layout orientation of its children, which is either vertical or horizontal, and the number of its children. We consider the layout orientation because it plays an important role in the visual effects. For a leaf node, we record its type and size. For example, an Android application's user interface can contain the following UI element types: Button, ImageButton, FloatingActionButton, AutoCompleteTextView, EditText, CheckBox, RadioButton, ToggleButton, Switch, Spinner, SeekBar, DatePicker, TimePicker, TextView, ImageView, VideoView, ProgressBar, and RatingBar, among others. We consider the size of widgets because perceptually users are most sensitive to the size of two objects. When the difference of sizes is large, it will affect users' judgment of similarity.

After obtaining a visual-representation tree, we decompose it into an ordered list of root-to-leaf paths, based on which the similarity of trees is calculated. The benefits are twofold: (1) Comparing the similarity at the sequence level provides the flexibility of identifying similar sub-regions. For user interface code, reusing the code of similar sub-regions is actually the most common use case. (2) Handling an ordered list of sequences is normally more computationally manageable. As proven by existing works, directly measuring tree similarity is not trivial due to the inherent complexity of trees. The decomposition can be done by a depth-first traversal of the tree. Preserving the order of the paths is critical because the order determines the relative positions of different widgets in the user interface.

B. CALCULATE THE SIMILARITY OF TWO SEQUENCES

Once we represent an input sketch or candidate code as an ordered list of root-to-leaf paths with sufficient visual information, we devise a novel solution to compare the similarity of two such lists. The building block is a method that calculates the similarity of two paths (or sequences), which we describe in this section. Sequence-level comparisons enable our solution to support inexact matches, which is a desirable property for code reuse.

Let the two ordered lists be $P = [p_1, p_2, \cdots, p_{|P|}]$ and $Q = [q_1, q_2, \cdots, q_{|Q|}]$, where p_i (q_j) is a sequence in P (Q), and $|P|$ ($|Q|$) is the number of sequences in P (Q). Let $|p_i|$ and $|q_j|$ be the numbers of nodes in p_i and q_j, respectively. We denote the lth node in p_i by p_i^l and the subsequence formed by the first l nodes by $p_i^{[1,l]}$. We divide the similarity of two sequences, p_i and q_j, denoted by $sim(p_i, q_j)$, into two parts. The first part is designed to capture the similarity of sequence layouts, which is denoted by $sim_l(p_i, q_j)$; the second part is designed to measure the widget similarity in terms of visual effects, which is denoted by $sim_w(p_i, q_j)$. The second part, in turn, is measured by the similarity of their leaf nodes' attributes because in a user interface the widgets represented by leaf nodes affect a user's visual perception most.

Now we are ready to present our novel method for comparing the similarity of two sequences for the purpose of user interface code retrieval. Our method is inspired by the *Levenshtein distance* [23], which is widely used for measuring the difference between two *strings*. However, our method differs from the Levenshtein distance in significant ways. In fact, in the experimental section, we show that a direct application of the Levenshtein distance results in subpar performance.

We give the pseudocode of our algorithm in Algorithm 1. For ease of presentation, we omit the subscripts of p_i and q_j. It takes as input two sequences p and q with their visual-related attributes and returns their similarity $sim(p, q)$ that is in the range of [0, 1]. In Lines 1–2, we reverse p and q to simplify the later operations. In Lines 3–22, we calculate the layout similarity. The idea is to compute the *layout distance* (`LD`) between two sequences in a dynamic programming fashion, based on which the layout similarity is computed. We first initiate a $|p| \times |q|$ matrix Φ which is filled with 0s. We make use of the following formula to iteratively update the value of each cell $\Phi(i, j)$.

$$\Phi(i,j) = \begin{cases} \max(i,j) - 1, & \text{if } \min(i,j) = 1 \\ \min \begin{cases} \Phi(i-1,j) + 1 \\ \Phi(i,j-1) + 1 \\ \Phi(i-1,j-1) + cost \end{cases}, & \text{otherwise} \end{cases} \quad (1)$$

The first row and first column of Φ are updated in Lines 4–11. After that, we update the remaining cells based on the concept of cost, which measures the additional cost required to change $p^{[1,i]}$ to $q^{[1,j]}$. The cost of insertion and deletion operations are assigned 1 because they will result in a structural change of the original trees and thus receive more penalties. The cost of substitution operations is more subtle. Its calculation requires considering the visual effects of p^i and q^j as well as those of p^{i-1} and q^{j-1}. We later explain how to calculate *cost* in Eq. 1 in an iteration in Algorithm 2. Once we obtain *cost* by invoking the `calCost` function in Line 14, we update the value of $\Phi(i, j)$ in Line 15. The layout distance `LD`(p, q), which is the value of $\Phi(|p|, |q|)$, is used to calculate the layout similarity $\text{sim}_l(p, q)$ (Line 21).

The widget similarity is calculated in Lines 22 and 23. Recall that a leaf node has two visual-related attributes: the

Algorithm 1 The Two Sequences Matching Algorithm

Input: Two sequences p and q
Output: sim(p, q)

1: $p \leftarrow$ reverse p;
2: $q \leftarrow$ reverse q;
3: Initialize a $|p| \times |q|$ matrix Φ filled with 0s;
4: **for** $1 \le i \le |p|$ **do**
5: $\quad \Phi(i, 1) = i - 1$;
6: $\quad i++$;
7: **end for**
8: **for** $1 \le j \le |q|$ **do**
9: $\quad \Phi(1, j) = j - 1$;
10: $\quad j++$;
11: **end for**
12: **for** $2 \le i \le |p|$ **do**
13: $\quad$ **for** $2 \le j \le |q|$ **do**
14: $\quad\quad cost = \texttt{calCost}(p^i, p^{i-1}, q^j, q^{j-1})$;
15: $\quad\quad \Phi(i, j) = \min(\Phi(i-1, j) + 1,$
$\quad\quad\quad\quad\quad\quad \Phi(i, j-1) + 1,$
$\quad\quad\quad\quad\quad\quad \Phi(i-1, j-1) + cost)$;
16: $\quad\quad j++$;
17: $\quad$ **end for**
18: $\quad i++$;
19: **end for**
20: $\texttt{LD}(p, q) = \Phi(|p|, |q|)$;
21: $sim_l(p, q) = 1 - \dfrac{\texttt{LD}(p, q)}{\max(|p|, |q|)}$;
22: $sim_{size}(p, q) = 1 - \left| \dfrac{size(p^1)}{size(p^{|p|})} - \dfrac{size(q^1)}{size(q^{|q|})} \right|$;
23: $sim_w(p, q) = \frac{1}{2} sim_{size}(p, q) + \frac{1}{2} sim_{type}(p, q)$;
24: $sim(p, q) = \omega_l sim_l(p, q) + \omega_w sim_w(p, q)$;
25: **return** $sim(p, q)$;

size and the type. Line 22 calculates the similarity between the leaf nodes of p and q in terms of their sizes. $size(\cdot)$ gives the size of a node. We consider the size because there has been empirical evidence to show that perceived size affects a user's performance in different tasks [30]. The similarity in terms of type is 1 if the two types are identical and 0 otherwise. The widget similarity is a weighted sum as calculated in Line 23. Finally, we calculate the similarity between p and q, $sim(p, q)$, in Line 24, where $\omega_l + \omega_w = 1$. We experimentally study the impact of different values of ω_l and ω_w in Section IV.

Now we explain the `calCost` function, which is given in Algorithm 2. $num(\cdot)$ gives a node's number of children and $ort(\cdot)$ returns a node's children's layout orientation. Recall that, for an internal node, there are two visual-related attributes, the layout orientation of its children and the number of its children. Intuitively, the cost is measured by the difference between these two attributes using three different scores $\{0, 0.5, 1\}$, where 0 indicates that both attributes "match", 0.5 indicates that one of the two attributes "matches" and 1 indicates none of them "match". Concretely, we start by handling two special cases. If the

Algorithm 2 The `Calcost` Function

Input: Four nodes p^i, p^{i-1}, q^j, q^{j-1}
Output: $cost$

1: **if** $num(p^i) = 1$ or $num(q^j) = 1$ **then**
2: $\quad$ **return** 0;
3: **end if**
4: **if** $ort(p^i) \neq ort(q^j)$ **then**
5: $\quad$ **return** 1;
6: **end if**
7: **if** $(ort(p^i) = ort(p^{i-1})) \oplus (ort(q^j) = ort(q^{j-1}))$ **then**
8: $\quad cost = 0.5$;
9: **else if** $ort(p^i) \neq ort(p^{i-1})$ and $ort(q^j) \neq ort(q^{j-1})$ **then**
10: $\quad$ **if** $num(p^i) = num(q^j)$ **then**
11: $\quad\quad cost = 0$;
12: $\quad$ **else**
13: $\quad\quad cost = 0.5$;
14: $\quad$ **end if**
15: **else**
16: $\quad cost = 0$;
17: **end if**
18: **return** $cost$;

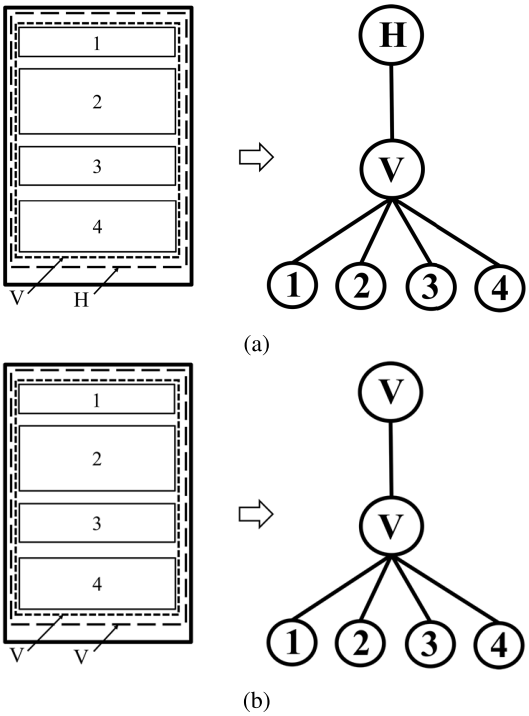

FIGURE 3. An example of two visual-representation trees in which different layout orientation values in the roots do not alter the visual representation. In general, when an internal node has only one child, the "V" label and "H" label are interchangeable.

number of children of p^i or q^j is 1, we return a cost of 0 (Lines 1–3). This is because if a node has only one child, different layout orientation values lead to the same visual effect. This is illustrated by Figure 3, where different orientation values in the roots do not affect the visual representation. Otherwise,

if the layout orientations of p^i and q^j are different, we return a cost of 1 because they result in substantially different visual representations. Lines 7–17 calculate the cost based on the visual effects of p^i and q^j on their children p^{i-1} and q^{j-1}. Depending on the orientations of p^i (q^j) and p^{i-1} (q^{j-1}), we assign different values to *cost*. In Line 7, $\oplus$ means XOR.

C. CALCULATE THE SIMILARITY OF TWO ORDERED LISTS OF SEQUENCES

In the previous section, we have presented the algorithm to calculate the similarity between two sequences. With the pairwise similarities, we design an algorithm to calculate the similarity between two ordered lists of sequences P and Q. The similarity of two user interfaces is calculated as a weighted sum of sub-region similarities. If two user interfaces have similar sub-regions, their similarity tends to be large. Intuitively, calculating the similarity of two ordered lists in terms of their visual representations requires to establish *1-to-1 correspondences* among the sequences while considering their positions in the user interfaces. Without a careful design, a naïve method might easily have a runtime complexity $O(\min(|Q|!, |P|!))$, where $|Q|$ and $|P|$ are the numbers of sequences in the two lists. In this paper, we present an algorithm that is of complexity $O(|Q| \cdot |P|)$. The general idea is to first calculate all pairwise similarities, then adjust their similarities based on their indexes (or order) in P and Q, and finally generate the similarity from the updated pairwise similarities. We give the pseudocode in Algorithm 3.

Without loss of generality, we assume that the number of sequences in P is larger than or equal to that in Q. We first initialize a $|P| \times |Q|$ matrix Φ. For a sequence pair p_i and q_j, we first calculate their similarity using Algorithm 1 (Line 4), and then adjust their similarity based on their orders in P and Q. The relative orders roughly determine their positions in the user interfaces. In Line 5, we introduce the factor ζ to quantify their similarity in terms of their orders.

More similar orders lead to a larger similarity. The updated similarity between p_i and q_j, $\zeta \cdot sim(p_i, q_j)$, is stored in $\Phi(i, j)$.

After obtaining the updated similarities of all sequence pairs, we identify the most similar sequence in P for each sequence in Q. To avoid many-to-one matching, we process the sequences in Q in order, retrieve the largest $\Phi(i, j)$ whose i has not been matched, and add $\frac{1}{|Q|} \cdot \Phi(i, j)$ to $sim(P, Q)$ (Lines 13–18). Finally, we adjust $sim(P, Q)$ by $\frac{|Q|}{|P|}$. This is to reflect the fact that a larger difference between the number of widgets in P and the number in Q makes the user interfaces less similar.

Algorithm 3 The Multi-Sequences Matching Algorithm

Input: Two ordered lists of sequences P and Q with $|P| \geq |Q|$
Output: $sim(P, Q)$

1: Initialize a $|P| \times |Q|$ matrix Φ filled with 0s;
2: **for** $1 \leq i \leq |P|$ **do**
3: **for** $1 \leq j \leq |Q|$ **do**
4: Calculate $sim(p_i, q_j)$ using Algorithm 1;
5: $\zeta = 1 - \left| \frac{i}{|P|} - \frac{j}{|Q|} \right|$;
6: $\Phi(i, j) = \zeta \cdot sim(p_i, q_j)$;
7: $j++$;
8: **end for**
9: $i++$;
10: **end for**
11: $sim(P, Q) = 0$;
12: $S_i = \emptyset$;
13: **for** $1 \leq j \leq |Q|$ **do**
14: Get the largest $\Phi(i, j)$ whose $i \notin S_i$;
15: Add i to S_i;
16: $sim(P, Q) += \frac{1}{|Q|} \cdot \Phi(i, j)$;
17: $j++$;
18: **end for**
19: $sim(P, Q) = \frac{|Q|}{|P|} \cdot sim(P, Q)$;
20: **return** $sim(P, Q)$;

D. DISCUSSION

Our user interface code retrieval solution is based on *static code*. In reality, rendering actual user interfaces with all dynamics of candidate code might help users make a better decision because the users are given the chance to observe how the real interfaces look like. Nevertheless, we deem that our approach still provides a reasonably good solution to user interface code retrieval for at least two reasons. First, static code of mobile applications is able to provide a large amount of information regarding actual dynamic interactions in the applications. For example, in static code there are event code snippets (e.g., click events of buttons) used to generate interactions. Second, while rendering real user interfaces generally helps users make a better decision for code reuse, sometimes the actual visual effects of user interfaces could be misleading. For example, consider an ad slot which displays different types of ads (e.g., videos, GIFs, or static images). Users might be misled by different creatives provided by advertisers to reckon that these are different UI elements.

Our solution is orthogonal to traditional keyword-based code retrieval methods. They can complement each other. For example, one can use a keyword-based method as a pre-filtering step to narrow down candidates and then apply our solution to identify the most relevant code. This will improve users' productivity.

It is worth noting that applying our solution requires to first generate visual-representation tree structures and maintain up-to-date tree structures. Essentially, this is the same as creating keyword indexes (e.g., inverted indexes) for keyword-based code retrieval methods. While the visual-representation tree structure is more complex than keywords, fortunately such structures can be generated in a highly efficient manner. It takes roughly 1 minute to generate

the tree structures of 100,000 user interface files. Since this process can be easily parallelized, we deem that our method is practical for real-world applications.

IV. EXPERIMENTAL EVALUATION

In this section, we conduct an experimental study to demonstrate that our visual-representation-aware tree matching algorithm outperforms the state-of-the-art algorithms. Since there is no well-established dataset for evaluating the performance of user interface code retrieval, we first carefully design an experiment to generate the similarity scores of different user interfaces based on real users' judgement. Based on this ground truth, we then show the superiority of our algorithm.

A. EXPERIMENTS FOR GENERATING GROUND TRUTH

Unlike the field of image retrieval, there does not exist any standard dataset which can be used to evaluate the performance of code retrieval, especially user interface code retrieval. To validate the effectiveness of our proposed method, we design an experiment to generate the similarity scores of 6,750 pairs of user interfaces based on users' manual evaluation and consider this dataset as the ground truth to subsequently evaluate different methods.

1) PARTICIPANTS

As the experiment requires a relatively long time to finish, all candidates were informed the estimated total time of the experiment and selected only if they can finish the experiment within 2 weeks. The final participants were paid to conduct the experiment. A total of fifteen Chinese graduate students (including 6 females and 9 males) with an average age around 28 were selected to participate in the experiment. All participants have computer science related background. They all have a deep understanding of software interface design and are experienced in Android software development. As such, they are able to provide professional judgement on the similarities of different user interfaces from the perspective of code reuse.

2) MATERIALS

To generate a representative ground truth dataset, we consider different categories of applications. From the app categories defined in Google Play and Apple App Store, we choose five mainstream categories, including social network, communication, education, multimedia, and news and magazines. We deliberately avoid the game category as game interfaces normally have disparate styles and therefore it is difficult to generate similar interfaces for participants to compare. For each of these five categories, we retrieve eight applications from GitHub and select 90 distinct user interfaces from them. These interfaces are of different types, including homepages, landing pages, and listing. As a result, a total of 40 applications and 450 user interfaces are selected. We screen the user interface pairs to obtain a more balanced similarity distribution, otherwise we might not be able to get sufficient similar pairs to validate our algorithm. The reason is that naturally the number of dissimilar pairs is much larger than that of similar pairs. Randomly sampling from all possible pairs will result in a highly skewed distribution, which cannot reliably validate our algorithm's performance on similar interfaces. After the preprocessing, a total of 6,750 user interface pairs are selected as the test set.

To avoid the influence of colors on participants' judgement, we decolorize all user interfaces. We also remind the participants that the purpose of this experiment is to evaluate the similarity in terms of code reuse, not merely the image similarity. For this reason, we also provide the source code of each interface to the participants.

3) PROCEDURE

In this experiment, we adopted the *within-subjects study design*, that is, each participant was required to evaluate all 6,750 interface pairs. We installed the experiment in the participants' laptops so as to provide them more flexibility. The interface pairs were organized into 30 groups based on their types, and each group was further divided into 15 rounds, each having 15 pairs evaluated. The participants need to finish rating a group at a time within 1 hour, and can work on different groups at different time based on their availability. This measure helps to prevent them from, for example, fatigue, which guarantees the reliability of the ratings they entered. All participants were required to conduct the experiment in a designated room so that the actual time they spent on the experiment could be verified, which helps to ensure the quality of their ratings.

The test platform illustrated in Figure 4 mimics the real code retrieval scenario, where a user inputs a sketch on the left and is rendered 15 candidate interfaces and their source code on the right. The participants can click an interface to see its source code. For each candidate interface, a participant is required to rate its reusability (or similarity) using a 1-5 point scale, where 5 means "extremely reusable" and 1 means "not at all reusable". To avoid judging the interfaces solely based on image similarities, the scales are given in terms of reusability instead of similarity.

4) ANALYSIS

We first show that all participants give highly consistent ratings. For each group of interface pairs, we measure the *intra-class correlation coefficient* (ICC) [31], which is a descriptive statistic used to measure how strongly users' ratings in the same group resemble those of others. The ICCs of ratings in different groups range from 0.725 to 0.889 (all p-values $<$ 0.0001), and the overall ICC is 0.822 (p-value $<$ 0.0001). This result indicates that all participants agree well on their judgement of reusability.

After the experiment, we obtain 15 ratings for each of the 6,750 pairs. Since the participants' ratings are highly consistent, we take the average of the 15 ratings as the final

FIGURE 4. The test interface used in our experiment.

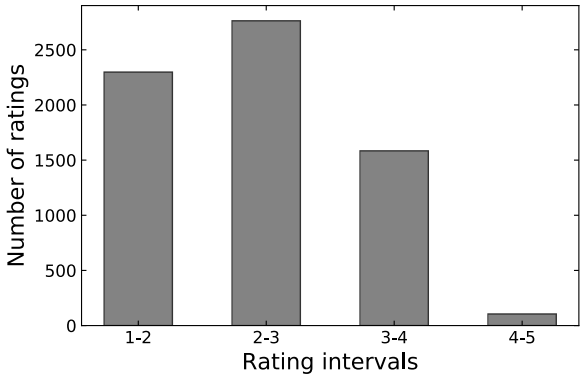

FIGURE 5. The distribution of final ratings of test interface pairs.

rating of a pair. That is, the final rating of pair i is

$$\bar{x}_i = \frac{\sum_j x_{ij}}{15},$$

where x_{ij} is the jth participant's rating of pair i. We plot the final rating distribution of all interface pairs in Figure 5. The ratings are discretized into four intervals: [1, 2), [2, 3), [3, 4), and [4, 5]. It can be observed that this test set contains sufficient similar interface pairs to validate our algorithm.

B. EXPERIMENTAL RESULTS

With the ground truth test set, we now evaluate the performance of our proposed solution.

1) EXPERIMENTAL SETTING

In practice, there are two types of widely-used sketches: pencil-on-paper sketches and pixel-based sketches [32]. Pixel-based sketches are basically digital bitmap images typically created by using Photoshop. Sometimes, pixel-based sketches could also come from a screenshot of an application. In general, the quality of pencil-on-paper sketches is difficult to control, making experimental results less representative. For this reason, we mainly consider pixel-based sketches in our experiments. Moreover, we treat the user interfaces of candidate code as approximations of pixel-based sketches. This is because: (1) pixel-based sketches are very close to actual user interfaces; (2) to generate statistically significant results, one needs a large number of sketches (e.g., thousands). This requires excessive efforts. Since generating a visual-representation tree from a sketch is not considered a technical contribution, we deem that this approximation is reasonable and that it can still effectively validate the performance of our solution. We note that for an input sketch, we generate its visual-representation tree using our variant of *pix2code* described in Section III-A, instead of parsing its code, which is able to create high-quality trees.

2) COMPETING METHODS

We compare our method with three state-of-the-art methods. The first is the rule-based method [1], [2] (referred to as `Rule`), which, to our best knowledge, is the only tree matching method for user interface code retrieval. Since the code of `Rule` is not publically available, we rewrite the algorithm in C# based on the description in [2]. Our implementation is available at [33]. The second is the classic tree edit distance algorithm [8][1] (referred to as `TED`), which is the most widely used general-purpose tree matching algorithm. The inputs to `TED` are the visual-representation trees of the sketch and candidate code. Since the original output of `TED` is the edit distance of two trees, we generate the similarity score as follows. Given two trees T_1 and T_2, the similarity score is

[1]The code is available at: https://github.com/timtadh/zhang-shasha

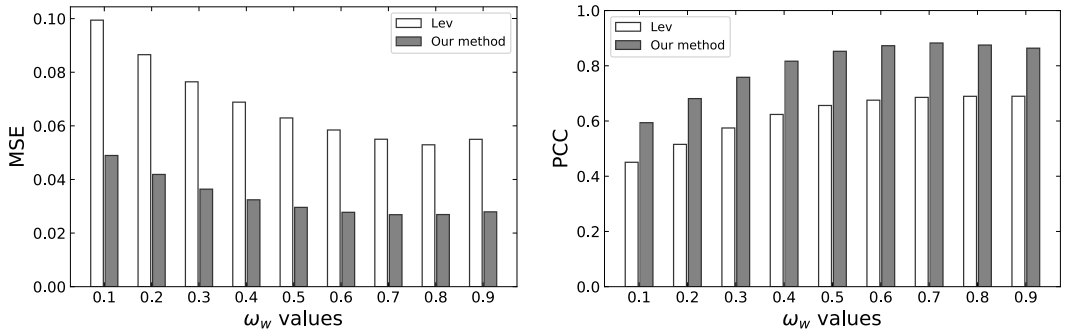

FIGURE 6. Performance of our method and `Lev` under varying ω_w values.

TABLE 1. The MSEs and PCCs of different methods.

Metric	TED	MDS	Rule	Our method
MSE	0.056	0.057	0.105	**0.027**
PCC	0.510	0.544	0.692	**0.883**

$1-\frac{d_e}{\max(|T_1|,|T_2|)}$, where d_e is the edit distance between T_1 and T_2, and $|\cdot|$ gives the number of nodes in a tree. The third is the algorithm in [28][2] (referred to as `MDS`), which represents trees as multidimensional sequences and measures their similarity by extracting various sequence structure features. Similarly, the inputs to `MDS` are the visual-representation trees of the sketch and candidate code.

In our experiments, we make use of two metrics to measure the performance of different methods. The first metric is the *mean squared error* (MSE). The MSE mainly evaluates the deviation between the similarity ratings given by different methods and the ground truth. The smaller the MSE, the better the performance. The second metric is the *Pearson correlation coefficient* (PCC). The PCC measures the degree of correlation between the similarity ratings given by a method and the ground truth. A PCC value of 1 is total positive linear correlation, 0 is no linear correlation, and -1 is total negative linear correlation. A larger PCC value indicates better performance.

3) PERFORMANCE COMPARISON WITH COMPETITORS

We present our major results in Table 1. Recall that our method has two weights ω_l and ω_w (with $\omega_l+\omega_w=1$) which control the relative importance of the layout similarity and widget similarity, respectively. Here we set $\omega_w=0.7$. In the next section, we study the impact of different weight values on the performance of our method.

As can be observed in Table 1, our method substantially outperforms the other three competitors in terms of both MSE and PCC. Our method reduces the smallest MSE of the competitors by 51.8% and improves the largest PCC of the competitors by 27.6%. `Rule` does not perform well in practice probably because the similarity is simply defined in terms of the visual containment relationship. As long as widgets of the same type are contained in the code tree, it positively contributes to the similarity score. However, this containment relationship based similarity rule is largely against users' visual perception. As a result, `Rule` tends to give overinflated ratings. A more systematic treatment is needed to properly leverage the visual information, as we did in our method. This explains why `Rule` has a much worse MSE.

`TED` and `MDS` cannot achieve desirable performance mainly because they largely ignore the visual information associated with a sketch or candidate code, which is essential for a programmer to judge the reusability of candidate code. It is intriguing to observe that, while being a much more complicated algorithm for measuring tree structure similarities, `MDS` does not bring much performance improvement. Its MSE is even worse than that of `TED`. We deem that this is an affirmative sign that leveraging only structure information cannot provide a satisfactory solution to user interface code retrieval. This fact justifies our contribution of proposing a visual-representation-aware approach.

4) IMPACT OF `LD` AND DIFFERENT WEIGHT VALUES

In Algorithm 1, when calculating the layout similarity of two sequences, we introduced the notion of *layout distance* (`LD`) which better measures the layout difference of user interfaces due to visual effects. In this section, we demonstrate the benefit of employing this distance measure by comparing it with the traditional Levenshtein distance. We construct a variant of our method in which we replace `LD` with the Levenshtein distance. We call this variant `Lev`. In addition, we are interested in understanding the impact of different weight values on the performance of our method and that of `Lev`. We plot the experimental results in Figure 6. The x-axis gives the weight values of widget similarity ω_w, ranging from 0.1 to 0.9; the y-axis in Figure 6(a) gives the MSEs of the two methods and the y-axis in Figure 6(b) gives the PCCs of the two methods.

[2]The code is available at: https://github.com/man1/Python-LCS

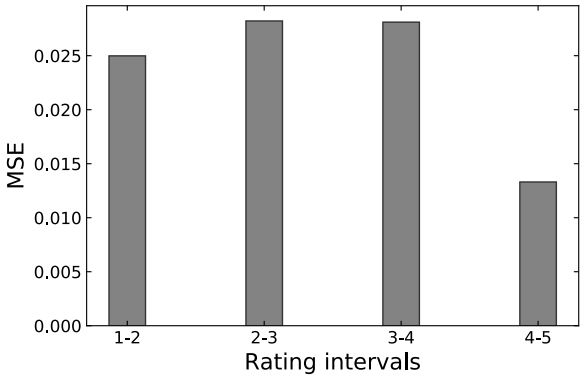

FIGURE 7. The MSEs of our method in different rating intervals.

We have three major observations. First, introducing LD indeed brings substantial performance improvement. Under different ω_w values, our method's performance is consistently much better than that of Lev. This suggests the importance of taking into consideration visual information and justifies the need of LD. Second, the weights play an important role in the resulting performance of our method and that of Lev. As ω_w increases, the performance of our method improves until $\omega_v = 0.7$. After that, the performance of our method starts to decline. This suggests that both layout similarity and widget similarity are important to the overall similarity and that widget similarity is relatively more beneficial. This is consistent with our previous analysis that widgets affect a user's visual perception most. It is interesting to observe that Lev achieves the best performance with a larger ω_w value (i.e., a value between 0.8 and 0.9). We deem that this is because Lev cannot obtain much visual information from the layout similarity and thus needs to rely more on the widget similarity. Therefore, the best performance of Lev is achieved with a larger weight on the widget similarity. These results prove that considering the visual effects indeed helps identify more reusable code. Finally, our method can achieve desirable performance in a relatively wide range of ω_w values (i.e., from 0.6 to 0.8). It follows that our method can be easily used in practice without much performance tuning on ω_w.

5) MSEs IN DIFFERENT RATING INTERVALS

In our last set of experiments, we study the MSEs of our method in different rating intervals. We note that, by the definition of PCC, studying the PCCs in each interval is less meaningful. We group the pairs into different intervals based on their ground truth ratings and report the MSEs of our method for each interval in Figure 7. This is to make sure that our method exhibits stable performance for interface pairs with different similarity levels. We can learn that our method is able to achieve small MSEs in all intervals. The MSEs in intervals [2, 3) and [3, 4) are slightly higher, but still smaller than 3%. This is natural because the boundary between these intervals is indeed more blurred, and it is thus more difficult to give an accurate rating. It is also interesting to observe that the MSE in interval [4, 5] is much smaller. This is probably because these pairs are indeed highly similar, allowing our method to generate very accurate ratings.

V. CONCLUSION AND FUTURE WORK

In this paper, we studied the problem of user interface code retrieval, which is the foundation of user interface code reuse. Despite its importance in user interface development, it was rarely researched in the literature. We analyzed the unique characteristics of user interface code and consequently proposed a novel visual-representation-aware solution. We proposed to represent an input sketch and candidate code as visual-representation trees to overcome the bottleneck of keyword-based search. We put forward two algorithms that effectively calculate the similarity between two visual-representation trees by fully leveraging various visual information. To validate our solution, we designed an experiment to generate a ground truth test set. Based on this test set, we showed that our solution outperforms three state-of-the-art methods.

There are a few interesting directions for future work. First, while we deem that our solution can provide reasonably good performance, it is worth exploring a more dynamic solution where users are rendered actual user interfaces generated from candidate code to better judge whether it is suitable for code reuse. Second, we plan to study how to extend our solution to more scenarios, for example, HTML code.

REFERENCES

[1] S. P. Reiss, "Seeking the user interface," in *Proc. ASE*, New York, NY, USA, 2014, pp. 103–114.
[2] S. P. Reiss, Y. Miao, and Q. Xin, "Seeking the user interface," *Automat. Softw. Eng.*, vol. 25, no. 1, pp. 157–193, 2018.
[3] A. Rauf, S. Anwar, M. A. Jaffer, and A. A. Shahid, "Automated GUI test coverage analysis using GA," in *Proc. ITNG*, Las Vegas, NV, USA, 2010, pp. 1057–1062.
[4] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. ASE*, Singapore, 2016, pp. 87–98.
[5] S. Wang, D. Lo, and L. Jiang, "AutoQuery: Automatic construction of dependency queries for code search," *Automat. Softw. Eng.*, vol. 23, no. 3, pp. 393–425, 2016.
[6] X. Tang, Y. Liang, X. Ma, Y. Lin, and D. Gao, "On the effectiveness of code-reuse-based Android application obfuscation," in *Proc. ICISC*, Seoul, South Korea, 2016, pp. 333–349.
[7] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.
[8] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM J. Comput.*, vol. 18, no. 6, pp. 1245–1262, 1989.
[9] P. Kilpeläinen and H. Mannila, "The tree inclusion problem," in *Proc. CAAP*, Brighton, U.K., 1991, pp. 202–214.
[10] P. Bille and I. L. Gørtz, "The tree inclusion problem: In linear space and faster," *ACM Trans. Algorithms*, vol. 7, no. 3, p. 38, 2011.
[11] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Trans. Softw. Eng.*, vol. 17, no. 8, pp. 800–813, Aug. 1991.
[12] W. B. Frakes and T. P. Pole, "An empirical study of representation methods for reusable software components," *IEEE Trans. Softw. Eng.*, vol. 20, no. 8, pp. 617–630, Aug. 1994.
[13] Y. Ye and G. Fischer, "Supporting reuse by delivering task-relevant and personalized information," in *Proc. ICSE*, Orlando, FL, USA, 2002, pp. 513–523.

[14] Y. Ye, "Programming with an intelligent agent," *IEEE Intell. Syst.*, vol. 18, no. 3, pp. 43–47, May 2003.
[15] S.-C. Chou and Y.-C. Chen, "Retrieving reusable components with variation points from software product lines," *Inf. Process. Lett.*, vol. 99, no. 3, pp. 106–110, 2006.
[16] S. Wang, D. Lo, and L. Jiang, "Code search via topic-enriched dependence graph matching," in *Proc. WCRE*, Limerick, Ireland, 2011, pp. 119–123.
[17] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School Comput.*, vol. 541, no. 115, pp. 64–68, 2007.
[18] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. ICSM*, Los Angeles, CA, USA, 1999, pp. 109–118.
[19] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, Mar. 2006.
[20] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. ICSE*, Minneapolis, MN, USA, 2007, pp. 96–105.
[21] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proc. ICSE*, Hyderabad, India, 2014, pp. 175–186.
[22] M.-M. Deza and E. Deza, *Dictionary of Distances*. Amsterdam, The Netherlands: Elsevier, 2006.
[23] V. I. Levenshtein, "Binary codes capable of correcting deletions," *Sov. Phys. Doklady*, vol. 10, no. 8, pp. 707–710, 1966.
[24] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *J. ACM*, vol. 24, no. 4, pp. 664–675, Oct. 1977.
[25] Y.-T. Tsai, "The constrained longest common subsequence problem," *Inf. Process. Lett.*, vol. 88, no. 4, pp. 173–176, 2003.
[26] H. Wang, "All common subsequences," in *Proc. IJCAI*, Hyderabad, India, 2007, pp. 635–640.
[27] J. Yang, Y. Xu, Y. Shang, and G. Chen, "A space-bounded anytime algorithm for the multiple longest common subsequence problem," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 11, pp. 2599–2609, Nov. 2014.
[28] Z. Lin, H. Wang, and S. McClean, "A multidimensional sequence approach to measuring tree similarity," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 2, pp. 197–208, Feb. 2012.
[29] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," in *Proc. EICS*, Paris, France, 2018, Art. no. 3.
[30] L. Goldfarb and J. Tzelgov, "Is size perception based on monocular distance cues computed automatically?" *Psychonomic Bull. Rev.*, vol. 12, no. 4, pp. 751–754, 2005.
[31] P. E. Shrout and J. L. Fleiss, "Intraclass correlations: Uses in assessing rater reliability," *Psychol. Bull.*, vol. 86, no. 2, p. 420, Mar. 1979.
[32] T. A. Nguyen and C. Csallner, "Reverse engineering mobile application user interfaces with REMAUI," in *Proc. ASE*, Singapore, 2016, pp. 248–259.
[33] *Visual-Representation-Aware Approach*. Accessed: Sep. 24, 2019. [Online]. Available: https://github.com/yingtao-xie/code_retrieval/

YINGTAO XIE received the B.Eng. and M.S. degrees in computer science from Sichuan University, China, in 2005 and 2010, respectively, where he is currently pursuing the Ph.D. degree with the College of Computer Science. He is currently an Associate Professor with China West Normal University. His research interests include human–computer interaction and software engineering.

TAO LIN received the M.S. degree in computer science from Sichuan University, China, in 2003, and the Ph.D. degree in information science from the University of Yamanashi, Japan, in 2007. He was with Waseda University, Japan, as a Visiting Lecturer. He is currently a Professor with Sichuan University.

HONGYAN XU received the master's degree in computer application technology from Shanghai Maritime University. He is currently pursuing the Ph.D. degree with Sichuan University, China. He is currently an Associate Professor and an Assistant Dean of the Institution of Intelligent Technology, Tianfu College, Southwestern University of Finance and Economics (SWUFE). His major is computer science and research interests include human–computer interaction and educational informationization.