2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

SolMiner: Mining Distinct Solutions in Programs

Lannan Luo
The Pennsylvania State University
University Park, PA 16802
lzl144@ist.psu.edu

Qiang Zeng
Temple University
Philadelphia, PA 19122
qzeng@temple.edu

ABSTRACT

Given a programming problem, because of a variety of data structures and algorithms that can be applied and different tradeoffs, such as space-time, to be considered, there may be many distinct solutions. By comparing his/her solution against others' and learning from the distinct solutions, a learner may quickly improve programming skills and gain experience in making trade-offs. Meanwhile, on the Internet many websites provide venues for programming practice and contests. Popular websites receive hundreds of thousands of submissions daily from novices as well as advanced learners. While these websites can automatically judge the correctness of a submission, none extracts distinct solutions from the submissions and provides them to learners. How to automatically identify distinct solutions from a large number of submissions is a challenging and unresolved problem. Due to diverse coding styles and high programming flexibility, submissions implementing the same solution may appear very different from each other; in addition, dealing with submissions at scale imposes extra challenges. We propose SolMiner, a solution miner, that automatically mines distinct solutions from a large number of submissions. SolMiner leverages static program analysis, data mining, and machine learning to automatically measure the similarity between submissions and identify distinct solutions. We have built a prototype of SolMiner and evaluated it. The evaluation shows that the technique is effective and efficient.

Keywords

Online judge, static program analysis, data mining, clustering

1. INTRODUCTION

Due to diverse coding experience and various viable designs, given a problem, people may come up with many distinct solutions [9, 26]. If these distinct solutions are provided to programming learners, they have chance to quickly improve programming skills and gain experience in creative designs by learning from the variety of solutions. For example, these solutions may demonstrate how different data structures and algorithms are applied and what trade-offs are considered. In addition, the diverse designs can inspire learners to solve problems more creatively in future coding.

In the meanwhile, on the Internet there are many online judge websites, such as CodeForces [6], TopCoder [33], CodeChef [5], and Hackerrank [10], which offer venues for programming practice and contests. Popular websites receive a large number of submissions. It can be very useful if we can extract distinct solutions from these submissions and provide them to learners as pedagogical resources. However, how to automatically identify distinct solutions is challenging. Due to diverse coding styles and high programming flexibility, submissions implementing the same solution may appear very different from each other, which makes it difficult to judge whether two seemingly different submissions have adopted the same solution. Moreover, dealing with submissions at scale imposes extra challenges.

Some initial efforts have been made to extract distinct solutions from submitted programs. Huang et al. proposed a technique that represents a program as a single abstract syntax tree (AST) and measures the similarity of code submissions by calculating the edit distance between the corresponding trees [13]. However, it is known that the edit distance problem for trees is NP-hard [1]. Although they use an approximation algorithm for tree comparison, it is still computationally expensive — quartic computational complexity for each pairwise comparison. Another work, OverCode, does not handle syntactic variability well; as a result, submissions using the same solution may be judged as distinct due to different coding styles [9]. How to extract distinct solutions from a large number of submissions efficiently and precisely is an unresolved problem.

We propose a novel technique, named SolMiner, for automatically mining distinct solutions from a large pool of submissions efficiently and precisely. Our observation is that static program analysis is effective in handling syntactic variability but is not scalable, while many data mining techniques are very scalable in processing large inputs. Thus, our idea is to combine techniques in the two areas to achieve both precision and scalability.

To this end, we first convert each code submission to a sequence of mini-ASTs, each of which represents a portion of a basic block in the control flow graph of the program. All the sequences are stored in a database. Then, data mining is applied to extracting characteristic mini-ASTs from the database, which are used as features in the clustering next. Through clustering, SolMiner builds a *dendrogram* (a tree diagram), such that submissions implementing the

ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA
 ISBN 978-1-4503-4205-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2889160.2889202

same solution are grouped into one cluster. Finally, distinct solutions can be extracted from these different clusters.

We have built a prototype of SolMiner and evaluated it using 100 coding problems in CodeForces; for each we downloaded all the submissions (up to over 18,000) from the website for our evaluation. We investigated (1) whether SolMiner can *precisely* separate submissions adopting distinct solutions into different clusters, and (2) whether SolMiner can mine solutions *efficiently*. The evaluation results show that SolMiner is precise and efficient.

In summary, we make the following contributions.

- We propose SolMiner that is able to mine distinct solutions from submitted code automatically. It is a novel approach that combines static program analysis, data mining, and unsupervised machine learning techniques to achieve both precision and scalability.
- We have implemented a prototype of SolMiner and evaluated it.

The rest of the paper is organized as follows. Section 2 discusses the motivation of our work, and Section 3 describes some background. Section 4 gives the problem statement. Section 5 presents the system design of SolMiner. We present the evaluation results in Section 6. The discussion and the related work are presented in Section 7 and 8, respectively, followed by a conclusion in Section 9.

2. MOTIVATION

We describe two different scenarios that have motivated our work.

A lot of programming learners use online judge systems (See Section 3.1) to practice programming and improve coding skills. In general, given a programming problem, there exist multiple solutions. By looking at different solutions, a learner can quickly learn how various algorithms and data structures can be applied. In addition, it can encourage learners to come up with novel and better solutions, which can effectively boost their learning interests and enhance their creativities and the abilities with innovation.

Another scenario is classroom teaching in programming courses. Consider a programming course taken by hundreds of students. Many programming assignments are given to the students during course. In the review classes of each assignment, the students may want to see the different solutions and compare their designs.

Both scenarios show that providing a variety of solutions to self-learners and students can be very useful. However, given a large number of submissions to a problem, it is impractical to manually identify distinct solutions, as it is a laborious, error-prone, and time-consuming process. Thus, it is beneficial if we can automatically discover and extract the distinct solutions from all the *accepted* programs, and collect these solutions to create an *editorial module*, which is a collection of distinct solutions provided to learners in order to facilitate programming education.

In the following, we use *program* and *submission* interchangeably, and refer them as an *accepted submission*, which contains a solution to the corresponding problem.

3. BACKGROUND

3.1 Online Judge

Online judge systems have become increasingly popular as automatic programming submission assessment platforms, offering invaluable venues for students, programming learners, and coding contestants [20, 3, 18]. An online judge system contains a repository of programming problems and is able to check the correctness of a code submission to a problem.

Some popular online judges in the Internet include UVa [35], Timus [32], Sphere [29], CodeForces [6], CodeChef [5], TopCoder [33], Hackerrank [10], etc. There are hundreds and thousands of learners or programmers submitting a large number of programs everyday, which contain many distinct solutions to the corresponding problems. Moreover, a few academic institutions have also implemented their own online judges serving as the automatic educational assessments for grading students' programming assignments. Examples include the *Scheme-Robo* system [27] implemented in the Helsinki University of Technology for automatically assessing programming exercises; the *PASS* system [4] developed in the City University of Hong Kong, which is a web-based automated programming assignment assessment system; and the *Jutge.org* [8], an open access educational online programming judge implemented in the Unversitat Politècnica de Catalunya.

Current online judge systems only give the correctness verdicts on the programs to learners [26, 14]. Possible verdicts include: *Accepted*, if the produced outputs of a program match the pre-specified answers exactly; *Wrong Answer*, if the produced outputs do not match the expected outputs; and *Execution Error*, if a program fails to produce outputs within the time/memory limit.

3.2 Frequent Itemset Mining

SolMiner applies a data mining technique, frequent itemset mining, which is described briefly in this section.

Frequent itemset mining is used in data mining tasks that try to find interesting patterns from databases [11]. Such a task can be stated as follows. Let $I = \{i_1, \cdots, i_m\}$ be a set of m unique items, and $D = \{t_1, \cdots, t_n\}$ be a database containing a set of n transactions. Each transaction in D is denoted as a tuple $\langle tid, S\rangle$, where tid is a unique transaction identifier, and S is an itemset that is a subset of I. Given an itemset $X = \{x_1, \cdots, x_k\}$, where $x_i \in I$ and $X \subseteq I$, the support of X is the number of transactions in D that contain X. Given a database D, frequent itemset mining is to determine all itemsets that are contained in at least a specified number of transactions. The specified number is called the *minmum support*, denoted as *min_sup*. X is said to be *frequent* if its support is greater than *min_sup*. Note that the items of X do not need to be contiguous or ordered when appearing in transactions that contain X.

To illustrate the concept, we take Table 1 as a simple example. In Table 1, the set of items is $I = \{a, b, c, d\}$, and a database D containing four transcations. Each transaction has a unique transaction identifier *tid* and an itemset X. If we set *min_sup* to 2, we obtain the following frequent itemsets and corresponding supports: $\{a:2\}$, $\{b:3\}$, $\{c:3\}$, $\{d:3\}$, $\{a,b:2\}$, $\{b,c:2\}$, $\{b,d:2\}$, and $\{c,d:2\}$ (the number following each itemset is its sup-

Table 1: A database *D*

tid	*S*
100	$\{a, b, c\}$
101	$\{b, c, d\}$
102	$\{a, b, d\}$
103	$\{c, d\}$

port), as each itemset is contained in at least $min\_sup = 2$ transactions.

4. PROBLEM STATEMENT

Given a problem, there may be multiple distinct solutions, which is illustrated by the following three simple examples. (1) A programmer is given a sequence of consecutive integers, and asked to compute the sum of these integers. A straightforward but slow method would be to add these integers one by one. A more efficient method would be to make use of the mathematical formula for summing up consecutive integers. (2) Given an unsorted list of elements, a programmer needs to put these elements in a certain order (e.g., numerical order or lexicographical order). Many sorting algorithms can be adopted to solve this problem, such as quick sort, merge sort, bubble sort, binary tree sort, etc. (3) A programmer is given a set of key-value pairs and asked to store them in a data structure, such that the value can be retrieved by providing a key. To solve it, there are several alternative data structures to choose from, such as list, red-black tree, and hash table. The three examples are very simple. However, most problems in online judge systems and programming courses are more complex; they usually consist of several subproblems. Each subproblem may also be solved by several distinct subsolutions. As a consequence, a solution to a problem is a combination of subsolutions. Two functionally equivalent code snippets, even if they use different programming coding styles, are regarded as the same (sub)solution.

Here, we give the definitions of a solution and a subsolution, as follows.

DEFINITION 1. *(Solution and Subsolution) Given a problem ρ consisting of n subproblems: $\{\rho_0, \ldots, \rho_n\}$. ρ is denoted as: $\rho = \bigcup_{i=0}^{n} \rho_i$. For each subproblem ρ_i, let $\xi_i = \{\epsilon_i^0, \ldots, \epsilon_i^m\}$ be a set of its distinct subsolutions.*

A subsolution to ρ_i is defined as:

$$\zeta(p_i) = \epsilon_i^j,$$

where $\epsilon_i^j \in \xi_i$. A solution to p is defined as:

$$\zeta(p) = \bigcup_{i=0}^{n} \zeta(p_i).$$

Given a programming problem and a set of its submissions, our goal is to separate these submissions into clear clusters, each containing the submissions that implement the same solution. However, because of various coding styles and high programming flexibility (e.g., the order of basic blocks or instructions) that lead to high variability, different programs implementing the same solution may be seemingly different from each other. A naive clustering implementation may probably assign them into different clusters. Therefore, we need to "peel off" these syntactic variations and capture the semantics of the submitted programs in order to cluster them correctly.

5. SYSTEM DESIGN

5.1 Overview

SolMiner converts the task of extracting distinct solutions to a data mining and clustering problem. First, SolMiner statically analyzes the source code of each program and represents a program as a sequence of mini-ASTs, each of which corresponds to a portion of a basic block in the control flow graph of the program. It collects all of these sequences to build a database; as a result, a set of programs is mapped into a database of sequences, each representing a program. Next, SolMiner extracts characteristic mini-ASTs, functioning as the features for clustering the programs. We call such characteristic mini-ASTs as *characteristic statements*, which capture the semantics of the programs. Finally, based on the characteristic statements, SolMiner separates the programs into different clusters; an editorial module is created by extracting all the distinct solutions from the clusters.

SolMiner comprises the following three functional parts: *parsing source code, extracting characteristic statements, and clustering programs.* Next, we discuss the design and implementation of each of the three functional parts. The current prototype of SolMiner works for programs written in Java, but the ideas described below should work in other programming languages.

5.2 Parsing Source Code

5.2.1 Process

To parse the source code, we need to get rid of the syntactic variations, and map functionally equivalent code to the same values. To achieve it, we adopt Soot [28], a manipulation and optimization framework for the Java programming language, to transform each program into an intermediate representation (IR). Specifically, we adopt *Jimple*, a 3-address IR, where each expression has explicitly declared and typed operands.

After transforming a program into the Jimple IR, SolMiner generates mini-ASTs for each basic block. A mini-AST here is a binary expression tree representing an expression, where the leaves are operands (i.e., constants or variable names) and the other nodes are operators. Each expression is a portion of a basic block in the program, which is a piece of straight-line code without any jumps or jump targets in the middle. To generate mini-ASTs for a basic block, SolMiner first aggregates the expressions into one if there are data flows between them. For example, the following two expressions, $a = b + c$ and $d = a + 1$, are aggregated into one expression $d = b + c + 1$. If an expression is a method call (e.g., $r = method_1()$), we do not aggregate it. After these, a mini-AST corresponding to an aggregated expression is generated; and each basic block is represented by one or more mini-ASTs.

Next, SolMiner removes the sequence numbers of the operands in each mini-AST, so variables of the same type are mapped into the same letter. In Jimple IR, different letters are used to represent different types; for instance, i represents an integer variable, l represents a long variable, d represents a double variable, etc. Finally, SolMiner computes a hash value for each mini-AST. It is worth nothing that two expressions with the same functionality may have different orders of operands and operators (e.g., $d = b * c + 1$ and $d = 1 + c * b$), resulting in different mini-ASTs. To handle it, SolMiner leverages the *commutative hash* method, which is recursively used to compute the hash value for each node from the bottom up based on the ASCII codes of its children nodes; the hash value obtained at the root node is used as the hash value of the mini-AST.

5.2.2 Algorithm

Algorithm 1 shows the pseudo-code for computing the hash value of a mini-AST. The input is the root node r of a

Algorithm 1 Compute a Hash Value for a Mini-AST

r: the root of a mini-AST

```
1: function HASH(r)
2:     if r is null then
3:         return 0
4:     else
5:         n = x = Hash(r → left) // get left child
6:         m = y = Hash(r → right) // get right child
7:         if r →value ∈ {+, ×, &, |, ∧} then
8:             n = max(x, y)
9:             m = min(x, y)
10:        end if
11:        return (r → value) * 379 + n * 541 + m * 73
12:    end if
13: end function
```

tree, where the leaves are operands and the other nodes are operators, and the output is the hash value of r.

`Hash` is a recursive function, which computes the hash value for a tree denoted by its root node r. It first checks whether r is null (Line 2). If so, 0 is returned (Line 3); otherwise, the hashes of the left child and the right child of the current node are first calculated (Line 5 and 6). If the current node is a *commutative* operator (Line 7), which means the order of operands does not affect the result of the expression, n is assigned as the bigger value of the left child's and right child's hash values, and m the smaller value of the two (Line 8 and 9). This way, a commutative expression is normalized. After that, a hash value for the current node is computed as shown in Line 11 (three prime numbers are used). The function effectively computes the hash value of each node in the tree from the bottom up, until reaching the root node; the hash value obtained at r is returned as the hash of the tree.

5.2.3 Example

Below is an example showing how to parse the source code and how to handle syntactic variations. The two code snippets separated by a horizontal line implement the same functionality in different ways.

```
m = n + 1;
tp = m * y[n];
ret = tp + ret;
```

```
res += ar[i] * (i + 1);
```

For both code snippet, we first utilize Soot to produce their Jimple IRs, and then aggregate these two IRs as follows.

```
i9 = ((i5 + 1) * r2[i5]) + i9;
```

```
i7 = i7 + (r2[i4] * (i4 + 1));
```

The sequence number of each operand is then removed, and a mini-AST is generated for each expression. Next, the commutative hash function is invoked to compute the hash values for the two mini-ASTs, and the same value is produced. Through the process, we "peel off" the implementation variations (e.g., different variable names, and different orders of instructions), and map functionally equivalent code to the same value.

5.3 Extracting Characteristic Statements

5.3.1 Process

After parsing each program into a set of hash values, the hash values of all the submissions to a given problem are stored into a database. SolMiner next extracts the *characteristic statements*, serving as the features in clustering. A statement here refers to an aggregated statement, which has been generated when computing hash values.

To find the characteristic statements, we need to exclude two kinds of statements. The first one is the *scarce* statements, contained in few programs and thus causing noises for clustering. For instance, if the `try-catch` statements are utilized in few programs, then they are filtered out (; however, if sufficient programs consider exception cases and adopt the `try-catch` statements, these statements will not be excluded). The second one is the *common* statements, which are included in the vast majority of the programs and thus do not only fail to differentiate the solutions but also slow down the clustering process. For example, most programs use the `system.out.println` statement, or the `readLine` statement, and hence these statements are excluded without further analysis.

We apply a well-known frequent itemset mining technique FP-Growth [11] to exclude the two kinds of statements. Two threshold minimum supports, min_sup_a and min_sup_b ($min\_sup_a < min\_sup_b$), are defined. A hash value of a statement here is analogous to an item in Section 3, and a sequence is analogous to a transaction. We scan the database and count up the *support* for each hash value. The support of a hash value is the number of sequences (i.e., programs) in the database that contain this hash value. If the support of a hash value is less than min_sup_a, it corresponds to a scarce statement. Similarly, if the support of a hash value is greater than min_sup_b, it corresponds to a common statement. The rest of the hash values correspond to the characteristic statements, which are used for clustering the programs. We discuss how to choose the values of min_sup_a and min_sup_b in evaluation (Section 6).

5.3.2 Example

Table 2: Database of seven programs. (a) is the initial database after the seven programs are parsed. (b) is the database of the same programs containing the characteristic statements only.

Program	Set of hash values
p_1	$\{1, 2, 3, 4\}$
p_2	$\{1, 2, 3, 4\}$
p_3	$\{1, 2, 3, 4, 8\}$
p_4	$\{2, 3, 4\}$
p_5	$\{4, 5, 6, 7\}$
p_6	$\{4, 5, 6, 7\}$
p_7	$\{4, 6, 7\}$

(a)

Program	Set of hash values
p_1	$\{1, 2, 3\}$
p_2	$\{1, 2, 3\}$
p_3	$\{1, 2, 3\}$
p_4	$\{2, 3\}$
p_5	$\{5, 6, 7\}$
p_6	$\{5, 6, 7\}$
p_7	$\{6, 7\}$

(b)

Table 2 illustrates an example of obtaining characteristic statements. There are seven programs, from p_1 to p_7, submitted for some coding problem. We first parse each program into a set of hash values and build a database, as showed in Table 2a. For the sake of presentation, we assume there are only eight unique statements with the hash values from 1 to 8. We first scan the database and count up the supports for each hash value. We obtain the following results: $\{1 : 3\}$, $\{2 : 4\}$, $\{3 : 4\}$, $\{4 : 7\}$, $\{5 : 2\}$, $\{6 : 3\}$, $\{7 : 3\}$, $\{8 : 1\}$ where the number after each colon is the support of the correspond-

ing hash value. Assume $min\_sup_a = 1$ and $min\_sup_b = 6$. Based on the thresholds, the hash value 8 corresponds to a scarce statement, and the hash value 4 corresponds to a common statement; the two kinds of statements are removed. The remaining statements are characteristic ones (as showed in Table 2b), which are then used in the clustering process.

5.4 Clustering Programs

5.4.1 Design Choice

There are various clustering techniques in the literature. In general, clustering techniques can be divided into two classes: partitional and hierarchical clustering. Partitional clustering directly decomposes objects into disjoint clusters, such that the intra-cluster similarity is maximized and the inter-cluster similarity is minimized [15, 12]. Famous partitional clustering methods include K-means and K-medoid methods. Hierarchical clustering [39], on the contrary, is a nested sequence of partitions; it proceeds successively to build a hierarchy of clusters. There are two strategies for hierarchical clustering. The first one is *bottom-up*, where each object forms a cluster on its own, and larger clusters are built by merging smaller ones; the second one is *top-down*, where all observations form in one cluster and splitting is performed recursively to divide the clusters into smaller ones. The result is a tree of clusters called a *dendrogram*, showing how the clusters are related.

We adopt the *top-down* hierarchical clustering because of the typical relation of multiple solutions. As aforementioned, a problem usually consists of several subproblems; each subproblem may be solved by several distinct subsolutions. Thus, a solution to a problem is a combination of the subsolutions. Therefore, the relation of solutions to a problem can be represented as a tree-like topological structure according to their subsolutions, which can be naturally analyzed by the *top-down* hierarchical clustering; this is elaborated below.

5.4.2 Multi-Solution Dendrogram

Given a database of programs, two different kinds of clustering are potentially useful. The first one is clustering statements, so that statements implementing the same subsolution are grouped together. The second one is clustering programs. We combine both of them and propose a new method which builds a *multi-solution dengrogram*, describing *how the programs are correlated in terms of the subsolutions*. The method recursively clusters the related statements using the frequent itemset mining technique, and then clusters the programs based on the clusters of the statements such that those implementing the same solution fall into the same cluster.

The process is iterative. In the beginning, all the programs are in one cluster. In each iteration of splitting, a frequent itemset, named *split itemset*, is selected to divide one cluster into two. The split itemset is selected as follows. First, we find all the frequent itemsets of hash values with *min_sup* as $\lambda \times N$, where $\lambda = 0.5$ initially and N is the total number of the programs. If no frequent itemset is returned, we reduce λ by 0.05 each time until we can find frequent itemsets; if $\lambda \leq 0.2$ and no frequent itemset is returned, which means the solutions do not have similarity, the splitting ends. Among the returned frequent itemsets, we exclude itemsets that have less than 4 items (i.e., statements). Again, the splitting ends if no frequent itemset remains. Among the remaining itemsets, we select the itemset with the highest support; if there is more than one itemset having the highest support, we select the one containing the largest number of items; if there are multiple itemsets having the highest support and containing the equally largest number of items, we randomly pick one working as the split itemset.

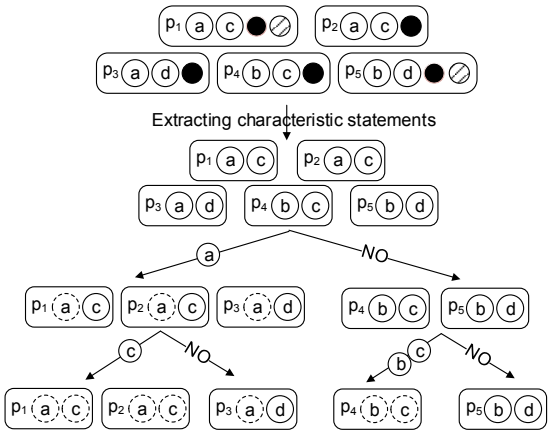

Figure 1: An example of building a multi-solution dendrogram to cluster programs. Each program implements two subsolutions, represented as circles. Subsolution a or b can solve the first subproblem. Subsolution c or d can solve the second one. A black circle represents a most-common statement. A shaded circle represents a non-related statement. A dotted circle represents the related statement are removed from the corresponding program at the current splitting iteration.

After selecting the split itemset, we separate solutions in a cluster into two smaller ones based on whether a solution contains the split itemset. For solutions that contain the split itemset, all the hash values in the split itemset are removed from each of the solutions. The cluster splitting then repeats on the two newly generated clusters, respectively. The splitting of a cluster ends if no split itemset can be found.

5.4.3 Example

As an example, Figure 1 shows how to cluster the programs. There are five programs, from p_1 to p_5, solving the same problem. The target problem has two subproblems, each of which has two distinct subsolutions: either a or b can solve the first subproblem, and either c or d can solve the second one. A black circle represents a common statement; for example, all the programs may utilize the `System.out.println` statement to output the results, and this statement is a common statement. A shaded circle stands for a scarce statement; for instance, only p_1 and p_5 may consider the exception case and adopt the `try-catch` statement, and this statement is a scarce statement. The common and scarce statements are excluded in the process of extracting characteristic statements.

After extracting characteristic statements, we cluster the programs by building a multi-solution dendrogram. We first find the frequent itemsets of hash values with *min_sup* of 0.5, and then select the split itemset. Assume the split itemset selected contains the hash values of the statements for the subsolution a (the assumption is for the simplicity of presentation; in practice the itemset does not have to cover all the hash values corresponding to a subsolution). Then we

use the selected split itemset to split the programs based on whether or not they contain this itemset. In this example, p_1, p_2 and p_3 are grouped into one cluster, while p_4 and p_5 form another. After this, all the hash values of the selected itemset are removed from p_1, p_2 and p_3 (demonstrated as dotted circles), and the splitting process continues. Finally, a multi-solution dendrogram that contains four clusters is built.

5.4.4 Algorithm

Algorithm 2 Programs Clustering

D: a database of many sequences of hash values

```
1: function ClusterPrograms(D)
2:     ExtractCharacteristicStatement(D)
3:     α ← Form(D) // all programs form in one cluster
4:     enq(α, Q) // insert α into queue Q
5:     while Q is not empty do
6:         β ← deq(Q)
7:         splitSet ← FindSplitItemset(α)
8:         if splitSet = null then
9:             continue
10:        end if
11:        (μ, ν) ← SplitCluster(β, splitSet)
12:        RemoveItemset(μ, splitSet)
13:        enq(μ, Q)
14:        enq(ν, Q)
15:    end while
16: end function
```

Algorithm 2 shows the pseudo-code for clustering programs. The input is a database D consisting of many sequences; each corresponds to a program. First, characteristic statements of each sequence are extracted (Line 2). The resulting sequences of D are then put in one cluster α, which is inserted into a queue Q (Lines 3 and 4). The loop then split each cluster in Q (Lines 5—15). For each cluster β, it first selects the split itemset (Line 7) and splits β into two clusters based on the split itemset (Line 11); the two generated clusters are then added into the queue (Lines 13 and 14) after the items in the split itemset are removed (Line 12). If the split itemset does not exist, the splitting continues with the next cluster in Q (Lines 8—10). The process returns until Q is empty.

5.4.5 Extracting Distinct Solutions

After clustering the programs, distinct solutions can be extracted by selecting a program from each cluster. We can randomly select one program from each cluster. Alternatively, the programs of each cluster can be packed into a program archive, and then presented to learners, such that learners have choices between programs for each solution. Moreover, leaf clusters in the dendrogram may be too many; in that case, a desired number of distinct solutions can be selected from intermediate nodes of some depth, which illustrates an advantaged of the tree topology of the clustering result.

6. EVALUATION

6.1 Experimental Settings and Dataset

To evaluate SolMiner, we crawled CodeForces [6], which is a popular online judge system dedicated to competitive programming. It consists of more than thousands of problems that anyone can participate to solve. We randomly selected 100 problems from CodeForces. For each problem, we obtained all the submitted programs, from which we selected the accepted ones written in Java. The current prototype of SolMiner works for programs written in Java, but it can be easily modified for working with other programming languages.

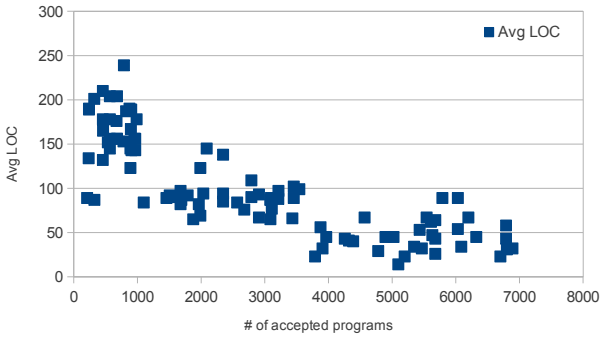

Figure 2: Distribution of 100 problems

We show the distribution of the 100 problems in terms of the number of the accepted programs and the average lines of code (LOC), in Figure 2. The x-axis represents the number of the accepted programs, and the y-axis stands for the average LOC of the accepted programs.

Table 3: Statistics of six problems.

Problem	Total prog.	Accepted prog. in Java	Avg LOC
p_1	10,625	4,256	43
p_2	11,750	3,875	56
p_3	7,650	2,034	94
p_4	8,325	1,963	82
p_5	6,750	965	186
p_6	4,650	534	203

Due to the space limit, 6 of the 100 problems have been randomly selected for detailed case description in the rest of the section. Their statistics are shown in Table 3, which includes the number of the total submitted programs, the number of the accepted programs written in Java, and the average LOC of the accepted programs.

Both Figure 2 and Table 3 show that the number of the accepted programs generally has a negative correlation with the average LOC of these programs. For example, the number of accepted programs of p_1 is larger than that of p_3, while the average LOC of p_1 is smaller than that of p_3. It is intuitive as the easier a problem, the smaller number of its solutions (i.e., the accepted programs), because more people have tried it and hence generated more accepted programs.

We evaluated SolMiner in the following two aspects: precision and efficiency. We seek to understand: (1) whether or not SolMiner can *precisely* cluster the programs into distinct solutions, and (2) whether or not SolMiner can handle a large number of programs *efficiently*. Our experiments were performed on a Linux machine with a Core2 Duo CPU and 8GB RAM.

6.2 Precision

To cluster a set of programs, SolMiner needs to extract characteristic statements by excluding two kinds of state-

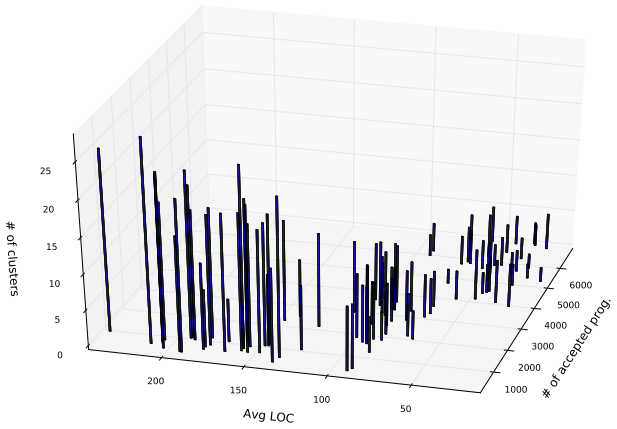

Figure 3: Clustering results for 100 problems

ments, scarce and common statements.

In our experiments, we set min_sup_a and min_sup_b as 0.1 and 0.9, respectively. That is, a code snippet that appears in less than 10% of the programs is not considered; as very few programs include it, it is not regarded as part of any representative solutions for being provided as a pedagogical example. On the other hand, a code snippet, such as the `System.out.println` statement, that appears in the majority of programs (>90%) is not a good candidate, as it does not provide characteristic information of a program. According to our experiments, 0.1 and 0.9 achieve a satisfactory clustering result. We can also dynamically adjust the threshold values based on different inputs, which will be studied in our future research.

6.2.1 Case Study

The entire clustering results for all the problems are showed in Figure 3. Table 4 shows the clustering results for the six problems. To demonstrate how SolMiner clusters the programs, we discuss clustering for 3 of the problems in detail.

Table 4: Clustering results for six problems.

Problem	# of clusters
p_1	3
p_2	4
p_3	9
p_4	6
p_5	13
p_6	11

Case 1. The first problem p_1 is to find a suitable placement for several chessmen (either white or black) on a given chessboard (some cells in the chessboard are bad and cannot be put a chessman), on the condition that no two chessmen with the same color are on two adjacent cells. For this problem, SolMiner found three distinct solutions. We show the characteristic statements with respect to the three solutions below.

The first and second code snippets contain characteristic statements with the same set of hash values, and submissions containing either of the two are clustered together. Because Soot transforms each program into an intermediate representation and the commutative hash method is utilized to compute the hash values, the syntactic variations are peeled off and functionally equivalent code snippets are regarded the same. The other two code snippets are recognized as characteristic statements of another two solutions, respectively.

```
//characteristic statements of the first solution.
if (chess[i][j] == '-') continue; // a bad cell
if ((j+i)%2 == 0)
    chess[i][j] = 'B'; // put a black chessman
else
    chess[i][j] = 'W'; // put a white chessman
```

```
//characteristic statements of the first solution.
if (map[i][j] != '-') // a good cell
     map[i][j] = (i+j)%2 == 0 ? 'B' : 'W';
```

```
//characteristic statements of the second solution.
if (ch[i][j] == '.')  // a good cell
    rek(i, j, 'B');    // rek is a recursive function

void rek(int i, int j, char h){
    ch[i][j] = h;
    char c = (h == 'B')? 'W' : 'B';
    if (i > 0 && ch[i-1][j] == '.') rek(i-1, j , c);
    if (i < n-1 && ch[i+1][j] == '.') rek(i+1, j, c);
    if (j > 0 && ch[i][j-1] == '.') rek(i, j-1, c);
    if (j < m-1 && ch[i][j+1] == '.') rek(i, j+1, c);
}
```

```
//characteristic statements of the third solution.
if (vec[i][j] == '.')   // a good cell
    pB(i, j);          // call pB

void pB(int i, int j) {
    vec[i][j] = 'B';
    if (i != vec.length-1)
        if (vec[i+1][j] == '.') pW(i+1, j); // call pW
    if (j != vec[0].length-1)
        if (vec[i][j+1] == '.') pW(i, j+1);
    if (i != 0)
        if (vec[i-1][j] == '.') pW(i-1, j);
    if (j != 0)
        if (vec[i][j-1] == '.') pW(i, j-1);
}
void pW(int i, int j) {
    vec[i][j]='W';
    if (i != vec.length-1)
        if (vec[i+1][j] == '.') pB(i+1, j); //call pB
    if (j != vec[0].length-1)
        if (vec[i][j+1]=='.') pB(i, j+1);
    if (i != 0)
        if (vec[i-1][j] == '.') pB(i-1, j);
    if (j != 0)
        if (vec[i][j-1] == '.') pB(i, j-1);
}
```

We compared the result to that of OverCode. Because OverCode does not handle implementation variations well, it determines the first and the second as two distinct solutions. In addition, for a common problem, OverCode usually generates too many clusters (usually over hundreds), which will confuse learners and fail to give them more explicit and clear programming directions. Although OverCode claims that clusters can be collapsed by rewriting rules, it is complex to write correct and appropriate rules manually.

Take the following two simple code snippets below as an example. They are classified as part of two different clusters (solutions) by OverCode, although the containing programs may implement the same solution. SolMiner regards them as equivalent.

```
c = a + b;  |  z = y * x;
z = x * y;  |  c = b + a;
```

Case 2. Consider another problem p_3 in Table 4. The task is to find a connected subgraph of a graph in a way that the density of the subgraph is as large as possible; the density is calculated using the nodes and edges. SolMiner found nine distinct solutions. Due to space limitation, we do not include the characteristic statements of each solution here. It is worth mentioning that SolMiner addresses the implementation variations caused by control flows. Take the following two code snippets (left and right) as an example.

```
for(int i=0;i<n;i++){
 int j=getNextX()%(i+1);
 int t=arr[i];
 arr[i]=arr[j];
 arr[j]=t;
}
```

```
int i=0;
while(i<n){
 int tmp=a[i];
 int x=getNextX()%(i+1);
 a[i]=a[x];
 a[x]=tmp;
 i++;
}
```

The left code snippet uses *for-loop*. The right one uses a *while-loop*. The two code snippets are functionality equivalent; the difference between them is caused by coding styles instead of using different algorithms. SolMiner successfully determined that the two code snippets belonged to the same solution. The IRs of the code snippets consist of three basic blocks respectively: (1) a block initializing the loop parameter, (2) a block checking the loop control condition, and (3) a block increasing the loop parameter and containing the loop body. Their IRs are similar. Moreover, as SolMiner adopts the frequent itemset mining technique, the order of the statements does not affect the result. Thus, the two code snippets are detected belonging to the same solution.

Case 3. The following two code snippets (left and right) are from programs solving p_6.

```
x = b1.get(i);
y = i - x;
z = a[y];
```

```
s = m.get(j);
t = n[s];
p = t - b;
```

The two code snippets are quite similar at the first glance. However, SolMiner detected them as distinct. The two code snippets correspond to different aggregated expressions, `z = a[i - b1.get(i)]` and `p = n[m.get(j)] - b`, respectively), which produce distinct hash values. This way, SolMiner correctly encodes the logic of the statements. Other problems manifest similar properties; due to the space limit, we skip the discussion here.

6.2.2 *Program Distribution*

Figure 4 shows the program distribution in different clusters for the six problems. For example, p_1 has three clusters (different colors represent different clusters), containing 2112 programs, 1645 programs, and 499 programs, respectively.

6.2.3 *Missing Splitting*

There are cases where further splitting is preferable but missing. When building a multi-solution dendrogram, SolMiner removes the hash values contained in the split itemset from each program at each splitting. On the other hand, the itemset corresponding to a subsolution is not considered in splitting, once it contains less than 4 hash values (Section 5.4.2). As a result, an itemset that should have been selected as the split itemset in later splitting is excluded from consideration because some of its hash values have been removed in earlier splitting. Take the following code snippets as an example.

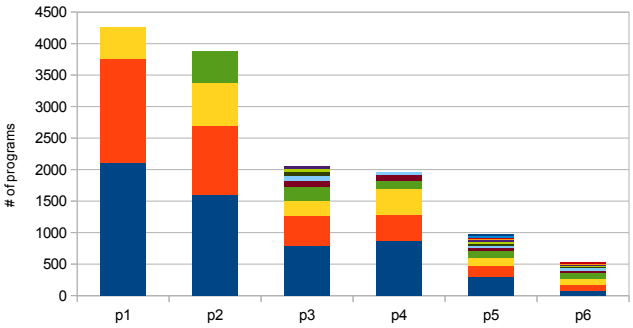

Figure 4: Program distribution in terms of the clusters for six problems

```
for(int i=0; i<n; ++i){
 sum += i;
}
```

```
for(int i=0; i<n; ++i){
 int s = m.get(i);
 t += n[s];
}
```

The two code snippets are part of two subsolutions (denoted as s_1 and s_2), respectively. Assume s_1 is chosen for splitting programs at a splitting iteration. After splitting, the statements contained in s 1 are removed from all the programs. As s_1 and s_2 share some hash values due to `for(int i=0; i<n; ++i)`, after the removal the number of hash values in s_2 becomes less than 4. So s_2, which may have been chosen as a split itemset in latter splitting, is not considered. This kind of cases leads to missing splitting, which means that programs implementing different solutions fall into the same cluster.

However, if programs only differ in terms of such a small code snippet, it is probable that they are different only due to a small subsolution and are identical in terms of other parts. Such missing splitting should not affect clustering quality much. On the contrary, by increasing the limit in terms of the smallest number of hash values in a subsolution (for example, changing it from 4 to 6), the algorithm can be configured to ignore difference between minor subsolutions, which will be investigated in future research.

6.3 Efficiency

We next study the efficiency of SolMiner. The running time of SolMiner for all the problems are between `3m16s` and `14m03s`. Therefore, SolMiner is efficient to be applied to cluster programs and discover distinct solutions in practice.

Time complexity. The time complexity of FP-Growth is $T = O(d^2n)$, where n is the number of programs in the database, and d is the number of unique items [19]. In our case, d equals the number of characteristic statements, which is correlated to the average LOC of the programs and does not grow as the number of submissions increases. This time complexity is for building one level of a multi-solution dendrogram, while the average height of the tree is $O(log(n))$. So the average time complexity of our algorithm is $O(d^2nlog(n))$. Thus, even for a large number of programs (e.g., a large class with many submissions), the time will not be long for discovering distinct solutions.

7. DISCUSSION

7.1 Precision Analysis

To cluster programs, we need to deal with implementation variations, which are manifested in two occasions: *intra-block* and *inter-block*. In this section, we discuss how SolMiner addresses them.

Intra-block variations. Because SolMiner transforms the source code of each program into the Jimple IR and then removes the sequence numbers of each operand in the IR, all identifiers of the same type are mapped into the same letter, regardless of their actual names. Moreover, SolMiner aggregates the expressions in a basic block into one expression; if there are data flows between them, the statements with the same functionality are mapped into the same value. Therefore, SolMiner is able to deal with the *intra-block* variations.

Inter-block variations. For the *inter-block* variations, they are mainly expressed in the following aspects: control flow and the order of basic blocks. In programming, there are different ways to implement control flows (i.e., branches and loops). Consider an *if-else* branch as an example. A programmer can either implement *if(a)* $block_1$ *else* $block_2$, or *if(!a)* $block_2$ *else* $block_1$. The two implementations have different branch conditions (i.e., a and $!a$) and different orders of the branch blocks (i.e., $block_1$ and $block_2$). Because we adopt the frequent itemset mining technique to extract characteristic statements, the order of the two branch blocks is irrelevant to our splitting. On the other hand, assume there is another program containing *if(b)* $block_i$ *else* $block_j$ which can solve the same problem, and neither $block_i$ nor $block_j$ is functionally equivalent with $block_1$ or $block_2$. To differentiate the two solutions, the branch blocks are sufficient. Similarly for *switch-case*, the order of the *case* branches does not affect clustering. Consider a loop control next. A programmer can either implement a *for-loop*, or a *while-loop*. SolMiner can also handle this, as analyzed and illustrated in our case study in evaluation. With respect to different orders of basic blocks, they are also taken care of by the frequent itemset mining technique, and have little impact on the result.

7.2 Future Work

The clustering based on a multi-solution dengrogram gives a good visualization on how the programs are correlated in terms of the distinct (sub)solutions. For example, in Figure 1, we can visualize that p_1 and p_2 implement the same solution, p_1, p_2 and p_3 share a subsolution, and p_4 and p_5 share another. At the current stage, we have not designed a user interface for visualizing the relation. In future, we will make an effort in this direction. Specifically, we will create an editorial module using distinct solutions and provide it to learners, to evaluate whether or not learners can actually learn something from looking at these distinct solutions. Moreover, we also sense that it is beneficial to automatically mine the common code segments from each cluster as the *core* solutions, which can provide more accurate and precise programming guidance to learners as pedagogical examples.

Furthermore, SolMiner can be extended to identify a new solution from a new program. After transforming the program into a sequence of hash values and extracting the characteristic statements, we can go through the multi-solution dengrogram in such a way that at each splitting point, the branch to be taken is determined by whether this sequence contains the split itemset previously chosen. If no cluster is reached at last, it implies that a new (sub)solution has been implemented. We plan to evaluate this in our future work.

8. RELATED WORK

8.1 Solution Identification

Several approaches that automatically identify solutions from a set of programs have been proposed in open literature. Taherkhani et al. proposed an instrument for identifying authentic students' sorting algorithm implementations [30]. It is based on static program analysis and machine learning methods. However, their approach can only be applied to identify the sorting algorithm. Our SolMiner can discover various algorithms. Huang et al. proposed a technique that represents a program as a single Abstract Syntax Tree (AST) and measures the similarity of code submissions by calculating the edit distance between the corresponding trees. However, it is known that the edit distance problem for trees is NP-hard [1]. Although an approximation algorithm for tree comparison is applied, for each pairwise comparison its time complexity is quartic in terms of the number of nodes in the trees [13]. Nguyen et al. proposed Codewebs, which creates an index of "code phrases" for all programs and semi-automatically identifies equivalence classes across these phrases [24]. Codewebs accepts queries that are subgraphs of an AST, and extracts all equivalent subtrees from the dataset. However, it does not handle coding variations well; consequently, programs implementing the same solution with different coding styles are regarded as distinct. Glassman et al. proposed OverCode, a system for visualizing and exploring thousands of programming solutions [9]. It uses both static and dynamic analysis to cluster similar solutions. However, its clustering algorithm does not handle coding variations well either.

8.2 Code Similarity Detection

Our work is also related to code similarity detection, or clone detection, aiming to find similar pieces of code from a code base. There is a substantial amount of work focusing on this problem, including string-based [2], AST-based [34, 38], token-based [17, 23, 25], and PDG-based [21, 7]. Some approached based on syntax analysis, such as bsdiff, bspatch, xdelta, JDiff, etc., are not effective in the presence of different coding styles and programming flexibility. Some other approaches based on dynamic analysis include API birthmark, system call birthmark, function call birthmark, and core-value birthmark. Tamada et al. proposed an API birthmark for Windows application [31]. Schuler et al. proposed a dynamic birthmark for Java. Wang et al. introduced two system call based birthmarks [36, 37]; however, they are not suitable for programs invoking insufficient system calls. Jhi et al. proposed a core-value based birthmark to measure program similarity [16]. However, it is not applicable when only partial code of a program is similar to another program. Luo et al. proposed CoP, a binary-oriented obfuscation-resilient method based on longest common subsequence of semantically equivalent basic blocks to measure the similarity between programs [22]. However, its computational overhead is high, and cannot be applied on a large number of programs.

9. CONCLUSION

We have presented *SolMiner*, which automatically mines distinct solutions from a set of programs. It is based on a novel approach that applies static program analysis, data mining, and machine learning. Online judge systems can utilize SolMiner to mine distinct solutions, which are invaluable

pedagogical examples to facilitate learning programming; and the classroom instructor can quickly generate an editorial module based on the submissions as well as the the distribution of submissions among distinct solutions. We have built a prototype of SolMiner. The evaluation results show that it is precise and efficient.

Acknowledgments

We would like to thank Dr. Shi Han and Dr. Dongmei Zhang from Microsoft Research China for their valuable suggestions and feedback, Dr. Peng Liu from Pennsylvania State University for his supporting the first author's work, and anonymous reviewers for their constructive comments.

10. REFERENCES

[1] T. Akutsu. Tree edit distance problems: Algorithms and applications to bioinformatics. In *IEICE transactions on information and systems*, 2010.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*, 1995.

[3] B. Cheang, A. Kurnia, A. Lim, and W.-C. Oon. On automated grading of programming assignments in an academic institution. *Computer & Education*, 2003.

[4] M. Choy, U. Nazir, C. Poon, and Y. Yu. Experiences in using an automated system for improving students' learning of computer programming. In *ICWL*, 2005.

[5] CodeChef. https://www.codechef.com/.

[6] CodeForces. http://codeforces.com/.

[7] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE*, 2008.

[8] O. Giménez, J. Petit, and S. Roura. Jutge.org: An educational programming judge. In *SIGCSE*, 2012.

[9] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller. OverCode: Visualizing variation in student solutions to programming problems at scale. *TOCHI*, 2015.

[10] Hackerrank. https://www.hackerrank.com/.

[11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM Sigmod Record*, 2000.

[12] A. Huang. Similarity measures for text document clustering. In *Proceedings of the New Zealand Computer Science Research Student Conference*, 2008.

[13] J. Huang, C. Piech, A. Nguyen, , and L. J. Guibas. Syntactic and functional variability of a million code submissions in a machine learning MOOC. In *MOOCshop*, 2013.

[14] P. Ihantola, T. Ahoniemi, and V. Karavirta. Review of recent systems for automatic assessment of programming assignments. In *Koli Calling*, 2010.

[15] A. Jain, M. Murty, and P. Flynn. Data clustering: A review. *ACM computing surveys (CSUR)*, 1999.

[16] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu. Value-based program characterization and its application to software plagiarism detection. In *ICSE*, 2011.

[17] J.-H. Ji, G. Woo, and H.-G. Cho. A source code linearization technique for detecting plagiarized programs. In *ITiCSE*, 2007.

[18] A. Kosowski and T. N. Michal Malafiejski. Application of an online judge & contester system in academic tuition. In *ICWL*, 2007.

[19] W. A. Kosters, W. Pijls, and V. Popova. Complexity analysis of depth first and fp-growth implementations of apriori. In *Machine Learning and Data Mining in Pattern Recognition*, 2003.

[20] A. Kurnia, A. Lim, and B. Cheang. Online judge. *Computer & Education*, 2001.

[21] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *KDD*, 2006.

[22] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *FSE*, 2014.

[23] MOSS: A System for Detecting Software Plagiarism. http://theory.stanford.edu/~aiken/moss/, 2013.

[24] A. Nguyen, C. Piech, J. Huang, and G. L. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World Wide Web*, 2014.

[25] L. Prechelt, G. Malpohl, and M. Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, Univ. of Karlsruhe, 2000.

[26] M. A. Revilla, S. Manzoor, and R. Liu. Competitive learning in informatics: The UVa online judge experience. In *Olympiads In Informatics*, 2008.

[27] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *ITiCSE*, 2001.

[28] Soot. http://sable.github.io/soot/.

[29] Sphere. http://www.spoj.com/.

[30] A. Taherkhani, A. Korhonen, and L. Malmi. Automatic recognition of students' sorting algorithm implementations in a data structures and algorithms course. In *Koli Calling*, 2012.

[31] H. Tamada, K. Okamoto, M. Nakamura, and A. Monden. Dynamic software birthmarks to detect the theft of Windows applications. In *ISFST*, 2004.

[32] Timus. http://acm.timus.ru/.

[33] TopCoder. https://www.topcoder.com/.

[34] N. Truong, P. Roe, and P. Bancroft. Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education*, 1991.

[35] UVa. https://uva.onlinejudge.org/.

[36] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In *CCS*, 2009.

[37] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *ACSAC*, 2009.

[38] W. Yang. Identifying syntactic differences between two programs. In *Software–Practice & Experience*, 1991.

[39] Y. Zhao and G. Karypis. Hierarchical clustering algorithms for document datasets. In *Data Mining and Knowledge Discovery*, 2005.