# CodeSimilarity ver. 2

Zirui Zhao
*University of Illinois at Urbana-Champaign*
ziruiz6@illinois.edu

Jonathan Osei-Owusu
*University of Illinois at Urbana-Champaign*
jo28@illinois.edu

## I. Abstract

Instructors of large introductory programming (CS1) courses are posed with the challenge of providing feedback to students at scale. Because these students have a diverse range of skills and backgrounds, it is important to be able to differentiate common approaches and shortcomings of a given problem. Having means to cluster submissions by similarity of approach grants instructors greater insight into providing for their courses. We present CodeSimilarity ver. 2, a technique for evaluating hundreds of correct code submissions and clustering them by approach taken. CodeSimilarity ver. 2 leverages automated test generation and path exploration to determine the symbolic return variables and path conditions of a given submission; comparing these values across all submissions allows us to establish behavioral equivalence relationships, with respect to the approach taken. Our experimental results demonstrate the effectiveness of our technique– by using symbolic return values and path conditions, our technique is able to cluster submissions having the same approach but different ASTs.

## II. Introduction

In large introductory programming (CS1) courses, instructors are posed with the challenge of providing feedback to students with incorrect submissions at scale. Programming exercises, in large introductory courses, can produce thousands of submissions. To provide customized feedback (i.e., partial credit), instructors have to inspect individual incorrect submissions to identify faults committed by students. For instructors, however, timely inspection of thousands of submissions is nearly an impossible task. For students, on the other hand, timely feedback is critical for their learning experience [1]. To maintain the quality of education while scaling to larger classrooms, There is a strong need to provide tool support for instructors and to provide scalable customized feedback to students.

To address this issue, we introduce CodeSimilarity ver. 2, a technique for the clustering of correct programming submissions based on their behavioral similarity, in terms of approach, to provide scalable customized partial grading to students. Our technique relies on symbolic return values and path conditions to form clusters.

CodeSimilarity ver. 1, the direct predecessor of this work, is a tool that allows student coding submissions from CS1 courses to be clustered based on their degrees of behavioral similarity, as determined by Paired Symbolic Execution (PSE). PSE is a technique implemented via the Pex symbolic execution engine to produce meaningful metrics approximating the dgrees of behavioral similarity between pairs of programs. The approach taken in CodeSimilarity ver. 1 gives insight into how behaviorally similar a given program is to another, in terms of producing the same outputs in response to given sets of inputs. CodeSimilarity ver. 1 is currently only able to form clusters based on a PSE-generated behavioral similarity metric, but it cannot differentiate different approaches taken in programming assignments (i.e., recursion versus iteration). This means that two submissions taking two different approaches may be classified as being behaviorally equivalent. The issue with this approach-blind clustering is revealed during hint generation– without a cluster of behaviorally equivalent submissions employing similar approaches, consistently providing relevant hints to a new submission is not possible. CodeSimilarity ver. 2 aims to go beyond its predecessor by forming clusters of submissions by approach taken. A concrete example of this is separating into two distinct clusters those submissions that employ recursive strategies from those that employ iterative ones. Clustering submissions by approach so will help us establish clusters at a finer granularity than would be achieved by simply evaluating input and output pairs. These more fine-grained clusters will allow instructors to more efficiently provide feedback to their students.

We believe CodeSimilarity ver. 2's ability to automatically cluster correct submissions by approach can be used in broad application to learning experience in introductory programming courses. For example, when a student needs help on a programming task, the student may not always get prompt feedback from instructors. It would be beneficial to automatically generate hints based on student's current submission. When students need help on task, these hints can be generated from other behaviorally equivalent submissions from other students who have already completed the programming exercise. Providing students with another level of support between them and instructors would help to greatly reduce the load placed on these instructors to manually inspect each submission.

## III. Approach & Implementation

In this section, we introduce our approach to cluster student submission approaches based on the path conditions and symbolic return-values that are collected from Pex. We first present an overview of our approach. We next show the technical details of the approach with an concrete example.

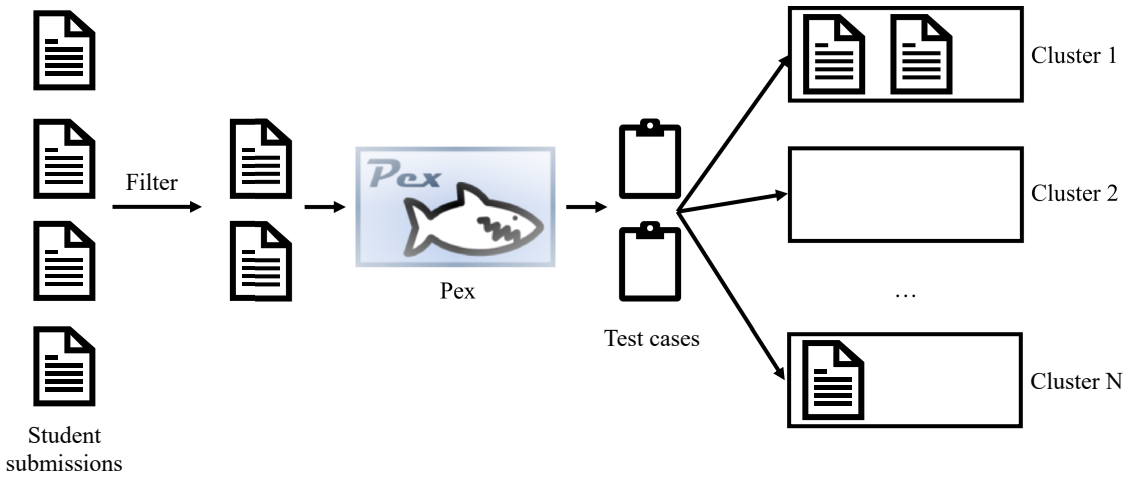


Fig. 1. The workflow of our approach.

### A. Overview

Traditional output-comparison based techniques are not sufficient to differentiate student submission approaches. Our techniques are based on the path-conditions and symbolic return-values comparison generated by Pex. For the same input, the path condition can reflect the student's approach and the symbolic return value can reflect the correctness of the implementation. Guided by this insight, our approach has the workflow that shows at Figure 1. Our approach has three stages. In the first stage, we pre-process student submissions to remove the source code that cannot be compiled or not written in C. In the second stage, we invoke Pex to run the same PUT on these student submissions and collect the test cases generated by Pex. For each test case, we also collect its corresponding path conditions and symbolic return value. In the third stage, we pick pairs of submissions and inspect their path conditions and symbolic return values. If their path conditions are equivalent and the symbolic return values are the same, then these two submissions are equivalent and belong to the same cluster. We write about 400 lines of Python for automated clustering and PUT creation.

### B. Constraint Comparison

To check the equivalence of path conditions and symbolic return values, we propose a string comparison based technique. Due to the limitation of Pex, we cannot easily get path conditions and symbolic return values in SMT-LIB format, which is accepted by Z3 or other SMT solvers. Also, if constraints contain external function calls, such as $Array.Sort$, it may cause too much overhead for SMT solvers to exam. Because these two reasons, we choose a string comparison based technique to check the equivalence of constraint models.

Our string comparison based technique has three stages. In the first stage, we pre-process the constraint model string. We remove all the formatting characters, such as `\n` and `\t`. We also remove irrelavent keywords and symbols generated by Pex, such as $return$ and semi-comas. After first stage, the string is a conjunction of several clauses. In the second stage, we split the string into a list of clauses and sort the list with alphabet order. In the third stage, we concatenate the elements in the list into a long string. After processing, if two constraints generate the same string, we consider these two constraints are equivalent.

The string comparison based technique is unsound and incomplete. If the string representation of the constraint contains redundant clauses, our technique will fail and cause false positive in clustering process. This problem can be mitigated by providing more rules to pre-process constraints.

### C. Clustering

For each student submission, we compare its path conditions and symbolic return values with one submission in each existing cluster. If two submissions have equivalent path conditions and the corresponding symbolic return values are equivalent, then the two submissions will be classified into the same cluster. Algorithm 1 and 2 describes our clustering technique.

**Algorithm 1:** Algorithm to compare two submissions.

```
1  Function same_cluster(submission1, submission2) is
       // initialize variables
2      test_cases1 ← copy(submission1.test_cases)
3      test_cases2 ← copy(submission2.test_cases)
4      for each t1 in submission1.test_cases do
5          found ← false
6          for each t2 in submission2.test_cases do
7              if t1.path_condition ==
                 t2.path_condition ∧ t1.ret_val == t2.ret_val
                 then
8                  found ← true test_cases1.remove(t1)
                   test_cases2.remove(t2)
9              end
10         end
11     end
12     return test_cases1.empty ∧ test_cases2.empty
13 end
```

**Algorithm 2:** Algorithm to cluster student submissions.

```
1  Function cluster(submissions) is
       // initialize variables
2      clusters ← map()
3      for each sub in submissions do
4          found ← false
5          for each c in clusters do
               // get the first element in the
                  cluster
6              cluster_sub ← clusters[c][0]
7              if same_cluster(cluster_sub, sub) then
8                  found ← true
9                  cluster[c].add(sub)
10             end
11         end
12         if ¬found then
               // not found, create a new cluster
13             cluster[sub] ← [sub]
14         end
15     end
16     return clusters
17 end
```

```
1   using System;
2
3   public class Program {
4     public static int Puzzle(int[] a) {
5       // student's code
6     }
7   }
```

Fig. 2. The template for student to implement.

## IV. Evaluation

In this section, we want to answer two research questions:

**RS1**: Can our approach cluster student submission approaches with a low false positive rate?

**RS2**: Can our approach get better clustering results compare to AST based approaches?

We use CodeHunt preview dataset for our evaluation. CodeHunt preview dataset contains around 13,000 submissions for a 48-hour worldwide contest. The contest has 4 sectors and each sector contains 6 puzzles. Because only a few of students complete puzzles in sector 3 and sector 4, and puzzles in sector 1 are too simple. We pick puzzles come from sector 2. Due to the Pex's weakness of handling string operations, we use problem *sector2-level5*, which does not contain string operations and also complex enough for different approaches, as the evaluation object. We filter all incorrect submissions and Java submissions. After filtering, we have 44 submissions left. We construct a Pex PUT template from the solution that is contained in the dataset. We use default settings for PUT.

The problem description of *sector2-level5* is *find maximum difference between 2 elements in an array*. Students need to implement the $puzzle$ method in Figure 2.

Figure 3 shows the clustering results and the number of submissions in each cluster.

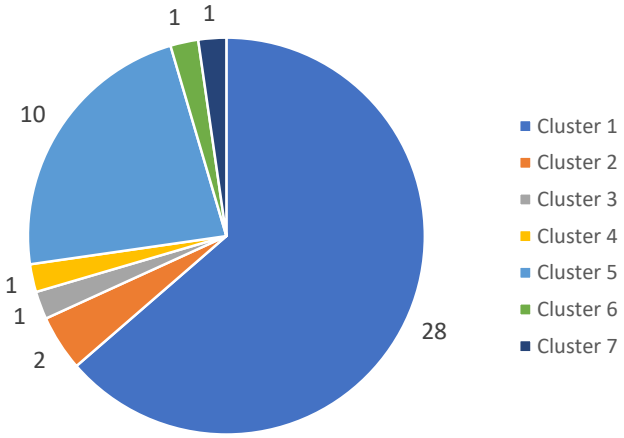


Fig. 3. Clustering results.

```
1   using System;
2   using System.Linq;
3   public class Program {
4     public static int Puzzle(int[] a) {
5       return a.Max()−a.Min();
6     }
7   }
```

Fig. 4. The template for student to implement.

For each cluster, we manually inspect and abstract its approach as the following:

**Cluster 1**: Calculate the maximum and the minimum element in the array then subtract them.

**Cluster 2**: Sort the array with $Array.Sort$ and iterate over the array to find the maximum difference.

**Cluster 3**: Sort the array with self-implemented bubble sort and subtract the first element from the last element.

**Cluster 4**: Enumerate all possible subtraction values.

**Cluster 5**: Sort the array with $Array.Sort$ and subtract the first element from the last element.

**Cluster 6**: An extra if-else structure then sort the array with self-implemented bubble sort and subtract the first element from the last element.

**Cluster 7** (false positive): Sort the array with $Array.Sort$ and iterate over the array to find the maximum difference.

Our approach generates 7 clusters for 44 student submissions. Only one cluster is redundant and it contains only 1 submission. So, our approach is good enough to differentiate student approaches with a low false negative rate.

To answer RS2, we manually inspect each cluster and compare the structure of submissions. For cluster 1, we find two popular structures. Figure 4 uses the method that is provided by the object to find the maximum and minimum element. Figure 5 uses a loop to find the maximum and minimum element. These two submissions have different ASTs but our approach still successfully put them in the same cluster.

```
1   using System;
2   public class Program {
3     public static int Puzzle(int[] a) {
4       int max = a[0], min = a[0];
5       for (int i = 0; i < a.Length; i++) {
6         if (max < a[i]) max = a[i];
7         if (min > a[i]) min = a[i];
8       }
9       return max - min;
10    }
11  }
```

Fig. 5. The template for student to implement.

## V. Threats to Validity

**Unsoundness.** Our technique is unsound, as Pex did have trouble generating symbolic return values and path conditions for some methods (i.e., C#'s StringBuilder object and ToCharArray method). This yielded false-positives in our results because Pex (1) lost mapping of symbolic variables and (2) concretized return vaariables. Moreoever, redundant path conditions also contributed to the number of false-positives; leveraging Z3 would help to decrease the number of observed false positives.

## VI. Related Work

Due to the growing nature of CS courses, clustering is an appealing approach to quickly understanding a large number of student submissions. A substantial body of work [2]–[7] exists to effectively cluster programming assignments based on program analysis. We next discuss three closely related approaches.

**Overcode.** Overcode [5] is a multi-stage approach of information visualization for assessing variations in correct submissions of programming exercises. It employs both dynamic and static analyses to cluster correct student submissions and applies visualization principles to show similarity and differences among the clusters. To conduct clustering, Overcode first uses dynamic analyses to produce *cleaned code*. It does so by detecting common and unique variables through variable renaming of program variables that share the same values in various traces across all students' submissions. Overcode then uses static analysis to group cleaned code submissions into a cluster if they contain syntactically identical program statements regardless of their order. Unlike Overcode, our clustering approach is designed for incorrect submissions. Overcode's technique of detecting common variables is more amenable to clustering correct student submissions because there is less variability in the values of the variables in correct submissions than in incorrect submissions.

**CLARA.** Similar to Overcode, CLARA [6] also clusters correct student submissions; however, instead of visualizing clusters, CLARA uses these clusters to generate program repairs for incorrect submissions. A cluster is defined based on a matching relation: two programs match if they have the same *control flow* and if there exists a *total bijective relation* between the variables of both programs. Two variables are said to be in such relation if they take the same values in the same order during the executions of the programs on the same inputs for all inputs in a given test suite. The clustering technique in CLARA imposes even stronger requirements. Program variables must have the same values in the same order across all program executions. In our setting of clustering incorrect submissions, this assumption is less likely to hold.

**MistakeBrowser.** MistakeBrowser [7] is a Mixed-Initiative program synthesis system for providing feedback to student submissions at scale. From the history of student submissions, MistakeBrowser first learns code transformations (code edits) for making an incorrect submission to be correct. Then, it uses these learned transformations to cluster incorrect submissions that share the same transformations. In contrast to our approach, we do not require pairs of incorrect and correct submissions but require a single correct solution (which can be a instructor-provided solution) and incorrect submissions to cluster.

## References


[1] S. A. Ambrose, M. W. Bridges, M. DiPietro, M. C. Lovett, and M. K. Norman, *How Learning Works: Seven Research-Based Principles for Smart Teaching*, 1st ed. Jossey-Bass, May 2010.

[2] S. Combéfis and A. Schils, "Automatic programming error class identification with code plagiarism-based clustering," in *Proceedings of the 2Nd International Code Hunt Workshop on Educational Software Engineering*, ser. CHESE 2016. New York, NY, USA: ACM, 2016, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2993270.2993271

[3] S. Sharma, P. Agarwal, P. Mor, and A. Karkare, "Tipsc: Tips and corrections for programming moocs," 04 2018.

[4] 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 *AIED Workshops*, 2013.

[5] 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," *ACM Trans. Comput.-Hum. Interact.*, vol. 22, no. 2, pp. 7:1–7:35, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699751

[6] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 465–480. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192387

[7] A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann, "Writing reusable code feedback at scale with mixed-initiative program synthesis," in *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, ser. L@S '17. New York, NY, USA: ACM, 2017, pp. 89–98. [Online]. Available: http://doi.acm.org/10.1145/3051457.3051467